# EvoKit

# EvoKit: EVOLUTIONARY COMPUTING FRAMEWORK MADE FOR RESEARCH

BY
YIDING LI, B.A.Sci

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSIITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ENGINEERING

| | |
|---|---|
| Master of Engineering (2024) | McMaster University |
| (Computing & Software) | Hamilton, Ontario, Canada |

| | |
|---|---|
| TITLE: | EvoKit: Evolutionary Computing Framework Made for Research |
| AUTHOR: | Yiding Li<br>B.A.Sci (Computer Science)<br>McMaster University, Hamilton, Canada |
| SUPERVISOR: | Dr. Stephen Kelly |
| NUMBER OF PAGES | 96 |
| DATE OF PRINTING | 2024-08-19 |

## Abstract

This project develops a framework to support research in evolutionary computing. The framework prioritises usability and extensibility, so that it is approachable to new users and can implement a diverse range of evolutionary search and optimization methods; this includes evolutionary algorithms and genetic programming.

Written from scratch in Python, the framework successfully implements two major paradigms of evolutionary computing [1]: genetic algorithms and genetic programming. Using these methods, the framework is able to solve OneMax and symbolic regression problems.

EvoKit is designed for novice and expert users alike. It is completely documented and provides tutorials for each major use case. This report adopts standardised terminologies from *Introduction to Evolutionary Computing* by Eiben and Smith [2]. The following materials supplement this report: the online repository, the online introduction, and the API documentation.

**Table of Contents**

## 1 Notations

These notations are used to describe components of algorithms.

TABLE 1: MATHEMATICAL NOTATIONS

| **Expression** | **Description** |
|---|---|
| $\mathcal{I}$ | Set of all individuals |
| $\mathcal{P}$ | Set of all populations |
| $a \in A$ | $a$ is in set $A$ |
| $P \in \mathcal{P}$ | $P$ is a population, same as $P \subseteq \mathcal{I}$ |
| $P \in \mathcal{P}_E$ | $P$ is a population of evaluated individuals |
| $P \in \mathcal{P}_U$ | $P$ is a population of unevaluated individuals |
| $x, x \in P$ | $x$ is an individual in population $P$ |
| $e \in E$ | $e$ is an evaluator |
| $s \in S$ | $s$ is a selector |
| $v \in V$ | $v$ is a variator |
| $a \circ b$ | Composition of functions $a$ and $b$; $a \circ b(x) = b\big(a(x)\big)$ |

The notations describing algorithms should be self-explanatory.

Table 2: Algorithm notations

| **Expression** | **Description** |
|---|---|
| $\iota \leftarrow b$ | Assign $b$ to identifier (variable) $\iota$ |
| obj.m | Member m of object obj |
| **global** $\iota_1$ : int | Declare identifier $\iota_1$ to represent a global variable |
| **access global** $\iota_1$ | Make global variable $\iota_1$ available in this scope |
| **destroy** $\iota_1$ | Destroy the association between $\iota_1$ and its value |
| *this is a comment* | A comment |
| *update a file*<br>*user_input*<br>*effect* | An operation or value that is not explicitly defined |
| uniform_sample(...) | A function that is not explicitly defined |

# 2 Introduction

Frameworks facilitate machine learning research. Examples such as Torch and TensorFlow streamline development, encourages interoperability of components, and facilitates the analysis of algorithms. Frameworks also support education, improving accessibility and helping non-experts to apply ML algorithms.

Paradigms of evolutionary computing originated as distinct fields of research. Since the term "evolutionary computing" was coined in the 1990s [1, p. 5], efforts have been made to unify these paradigms. Some examples are: (a) a textbook that teaches them as a single subject [2], (b) a theoretical model that describes them with the same language [3], and (c) frameworks that implement several algorithms using the same system [4] [5] [6]. EvoKit follows this trend, prioritising usability, understandability, and flexibility. These priorities are codified in section 4.2 Non-Functional Requirements as requirements. Sections 5.1 Design Decisions and 5.2 Module Design enumerate, respectively, modules and decisions that derive from these requirements.

The rest of this report is organised as follows:

- ◊ Section 2 outlines the problem models of evolutionary computing.
- ◊ Section 3 details goals and requirements of the framework.
- ◊ Section 4 lists design decisions and modules.
- ◊ Section 5 gives examples of applying the framework to solve several well-known problems.
- ◊ Section 6 collects other works that are referenced in this report.
- ◊ Section 7 forecasts possible ways to improve the framework.

## 3 Problem Model

Development of any problem-solving software must use a fixed understanding of the **problem model**. This section captures a description of evolutionary computing.

evolutionary computing

**Evolutionary Computing** (**EC**, also **evolutionary computation**) is a metaheuristic, iterative, stochastic, population-based search and optimisation method that emulates Darwinian natural selection. The main paradigms of evolutionary computing are: (a) *genetic algorithms*, (b) *evolutionary strategies*, (c) *evolutionary programming*, and (d) *genetic programming* [1].

Evolutionary computing has seen widespread use [7]. Some examples are: design of antennae [8], design of neural networks [9], evolving machine learning algorithms [10], and the discovery of physical laws [11]. Since 2004, the Humies award has been given to human-competitive results produced by evolutionary computing, totalling 50 gold/silver awards up to the 21st competition [12]. An article by Koza discusses trends of human-competitive results by genetic programming [13].

## 3.1 Components of an Algorithm

evolutionary algorithms

**Evolutionary Algorithms** (**EAs**) typically follow a fixed structure. A minimal iteration (or *generation*) of an evolutionary algorithm (EA) consists of the following steps [2, pp. 28-34], as shown in Figure 1.

population; individual

- The algorithm begins with a **population**: a collection of **individuals**. Each individual is a candidate solution to the problem.

variator

- The **variator** produces new individuals from existing ones.

evaluator

fitness

- The **evaluator** assigns a **fitness** to each individual, correlating to the individual's ability to solve the given problem. The fitness can be a real number or, in a muti-objective setting, a real vector.

selector

- The **selector** picks a subset of these individuals to replace ones in the population. *Generational algorithms* replace the entire population; *steady-state algorithms* replace a subset of the population.

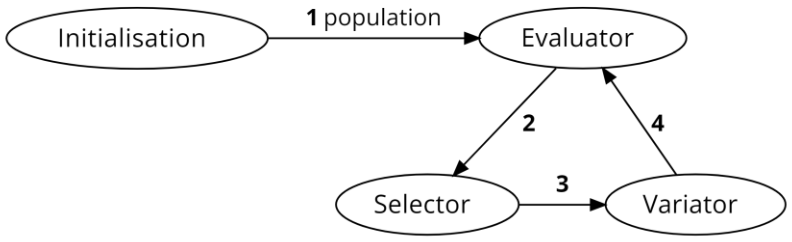

*Figure 1: Steps of an evolutionary algorithm*

search and optimisation

operator

As illustrated in Figure 2, this algorithm captures a **search and optimisation** process. In it, individuals (in the population) are a subset of the solution space; the evaluator captures how well the solution performs in the problem space, and the variator explores the solution space. The selector uses information given by the evaluator to steer exploration of the solution space (by selecting from the output of the variator). The evaluator, variator, and selector are **operator**s that act on the population.

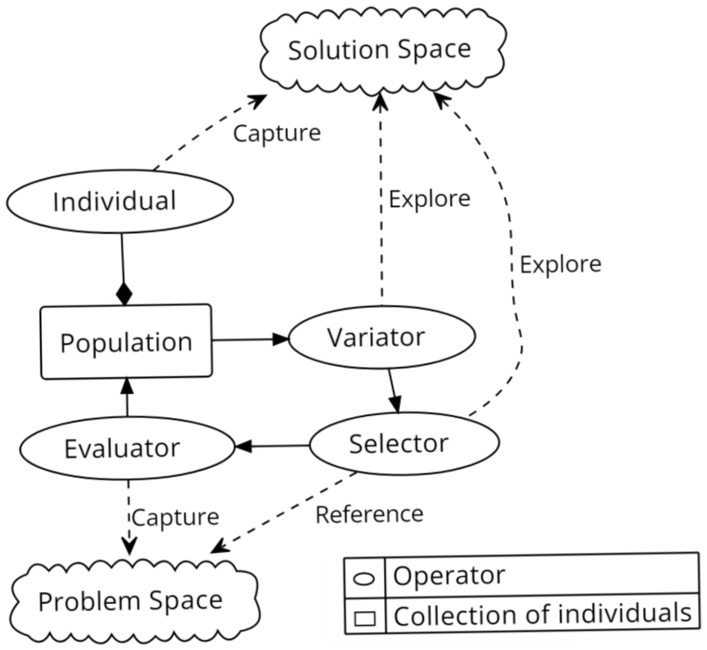

*Figure 2: A minimal evolutionary algorithm*

### 3.1.1 A Mathematical Formulation

An informal mathematical description of the model follows. Section 1 Notations collects all mathematical notations used in this document.

evaluated induvial

Each population is a set of individuals. An **evaluated individual** is an individual whose fitness attribute is set. For simplicity, either all individuals in a population is evaluated, or none of them are.

TABLE 3: NOTATIONS OF COLLECTIONS

| **Collection** | **Notation** | |
|---|---|---|
| Population | $P \in \mathcal{P}$, | $\mathcal{P} = \{ P' \mid P' \subseteq \mathcal{I} \}$ |
| Evaluated population | $P \in \mathcal{P}_E$, | $\mathcal{P}_E = \{ P' \in \mathcal{P} \mid i.\text{fitness defined } \forall i \in P' \}$ |
| Unevaluated population | $P \in \mathcal{P}_U$, | $\mathcal{P}_U = \mathcal{P} \setminus \mathcal{P}_E$ |

Operators act on populations. An evaluator maps a population to an evaluated population. A selector selects from an evaluated population to a subset of it. A variator maps from a population to an unevaluated population.

Table 4: Notations of operators

| **Operator** | **Notation** |
|---|---|
| Evaluator | $e \in E, \quad e' \subseteq \mathcal{P} \to \mathcal{P}_E \quad \forall\ e' \in E$ |
| Selector | $s \in S, \quad s' \in \mathcal{P}_E \to \mathcal{P}'_E \ \forall\ s' \in S, \mathcal{P}'_E \subseteq \mathcal{P}_E$ |
| Variator | $v \in V, \quad v' \in \mathcal{P} \to \mathcal{P}_U \quad \forall\ v' \in V$ |

### 3.1.2 A Variant Model

Evolutionary algorithms often select twice, once before and once after applying the variator [2, pp. 28-34].

The first selection (with the **parent evaluator** and the **parent selector**) [2, p. 31] select for high-fitness individuals to undergo variation, so that their advantageous traits are passed on. The second selection (with the **offspring evaluator** and the **survivor selector**) maintains the population size, to keep the cost of storing and evaluating individuals low [2, pp. 33-34]. Figure 3 visualises this structure:

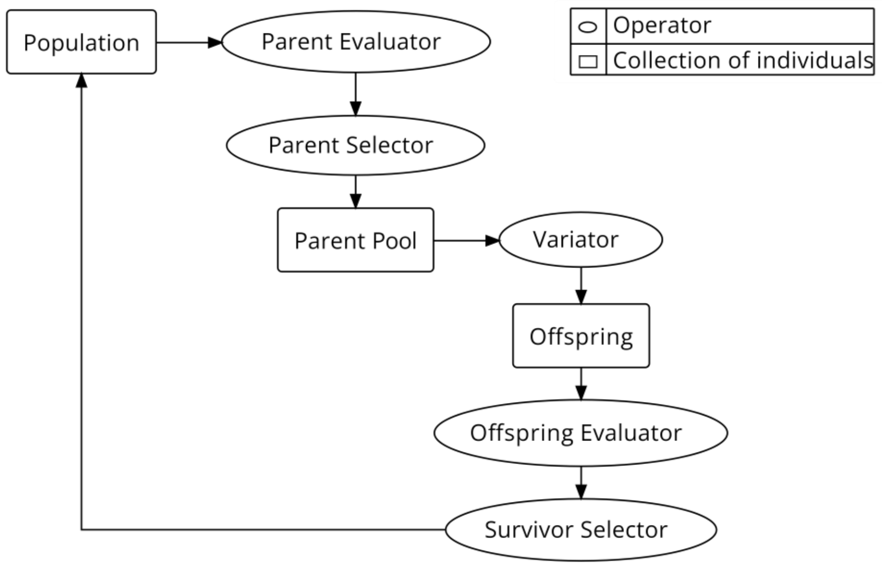

*Figure 3: Structure a typical evolutionary algorithm*

## 4 Requirements Specifications

primary goal

subgoal

As an evolutionary learning framework, EvoKit must facilitate the research and development of algorithms. This **primary goal** divides into three **subgoal**s: (a) the framework must be able to implement algorithms, (b) these algorithms must use interoperating operators, and (c) the framework must be able to analyse these operators.

requirement

Each subgoal decomposes to **requirement**s; each requirement inspires design and provide verifiable ways to achieve related goals. Conversely, all design decisions and development efforts must contribute to the satisfaction of requirements; all requirements must contribute to the satisfaction of goals. This approach (Figure 4) increases productivity by reducing wasted effort [14].

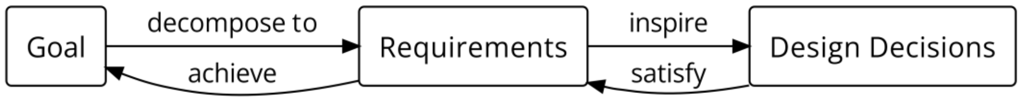

*Figure 4: Relation between goals, requirements, and design decisions*

## 4.1 Functional Requirements

functional requirement

**Functional requirement**s follow from goals, further detailing tasks that the software must complete. EvoKit's subgoals and functional requirements are as follows:

- GOAL 1: Provide a framework for implementing novel evolutionary algorithms.

  «Implement Algorithms» FREQ 1 ← (GOAL 1): The framework must be able to implement all evolutionary algorithms in Appendix 1.

- GOAL 2: Provide a framework that supports interoperation of evolutionary operators.

  «Interoperable Operators» FREQ 2 ← (GOAL 2): Ensure that operators of the same type should share one common behaviour so that they can be used interchangeably.

- GOAL 3: Facilitate the analysis and communication of evolutionary algorithms.

  «Visualise Algorithms» FREQ 3 ← (GOAL 3): Provide a way to visualise the learning process.

«Visualise Individuals» FREQ 4 ← (GOAL 3): Provide a way to visualise individuals.

## 4.2 Non-Functional Requirements

non-functional requirement

While functional requirements define tasks that the software must complete, **non-functional requirement**s describe qualities that the software must exhibit. That is, non-functional requirements prescribe *qualities*, instead of *behaviours* [15, pp. 45-71].

### 4.2.1 Usability

The ease of a use of a software must be defined with respect to user groups [16, p. 22]. EvoKit in particular should be usable to both novice and expert users.

abstraction

«Right Level of Abstraction» NFREQ 1: A powerful tool against complexity, **abstraction** separates the important from the inconsequential [16, p. 49]. The framework abstracts away common and repetitive tasks, so that its user can focus on developing novel algorithms.

«Self-Sufficient Instructions» NFREQ 2: The framework sufficiently and completely describes how it should be used. That is, a user can use all features of the software without consulting an external source.

«Ease to Learn» NFREQ 3: Without prior experience, a user can easily learn to use most features of the framework.

### 4.2.2 Understandability

For research software, understandability allows the researcher to (a) explain how the code leads to research findings and (b) reason that the code behaves exactly as described [17]. Understanding a software is also critical to its maintenance [16, p. 28].

«Transparency» NFREQ 4: The framework makes its inner workings visible to the user.

«Independence» NFREQ 5: The framework does not use any external module. Because external modules are maintained with the framework, they can have unexpected behaviours as either errors or undocumented features; the framework should avoid using external modules for this reason.

«Reproducibility» NFREQ 6: All sources of randomness can be controlled. When randomness is controlled, the same sequence of actions always lead to the same output.

### 4.2.3 Flexibility

"Many in the scientific computing software community believe that upfront requirements are impossible, or at least infeasible." [18, p. 1] Machine learning frameworks are especially vulnerable to this, because users in the future could use the framework to implement algorithms not known by today's researchers. The framework must anticipate this need.

«Modifiability» NFREQ 7: Users and maintainers of the framework can modify it to account for new requirements, in particular to add new capabilities.

«Portability» NFREQ 8: The framework operates normally in many different operating environments. This requirement is complementary to «Independence» NFREQ 5, which implies portability with respect to different software environments.

## 5 Design

This section lists design decisions and modules, each associated to requirements that motivate it.

traceability diagram

**Traceability diagram**s (Figure 5, Figure 6) trace from requirements to goals, and from design decisions and modules to requirements [19]. The diagrams serve two purposes:

1. By linking each decision to a requirement and each requirement to a goal, the diagram gives confidence that all development efforts are necessary.
2. The diagram shows how each requirement affects the program. When a requirement changes, only components that are affected by it need to change.

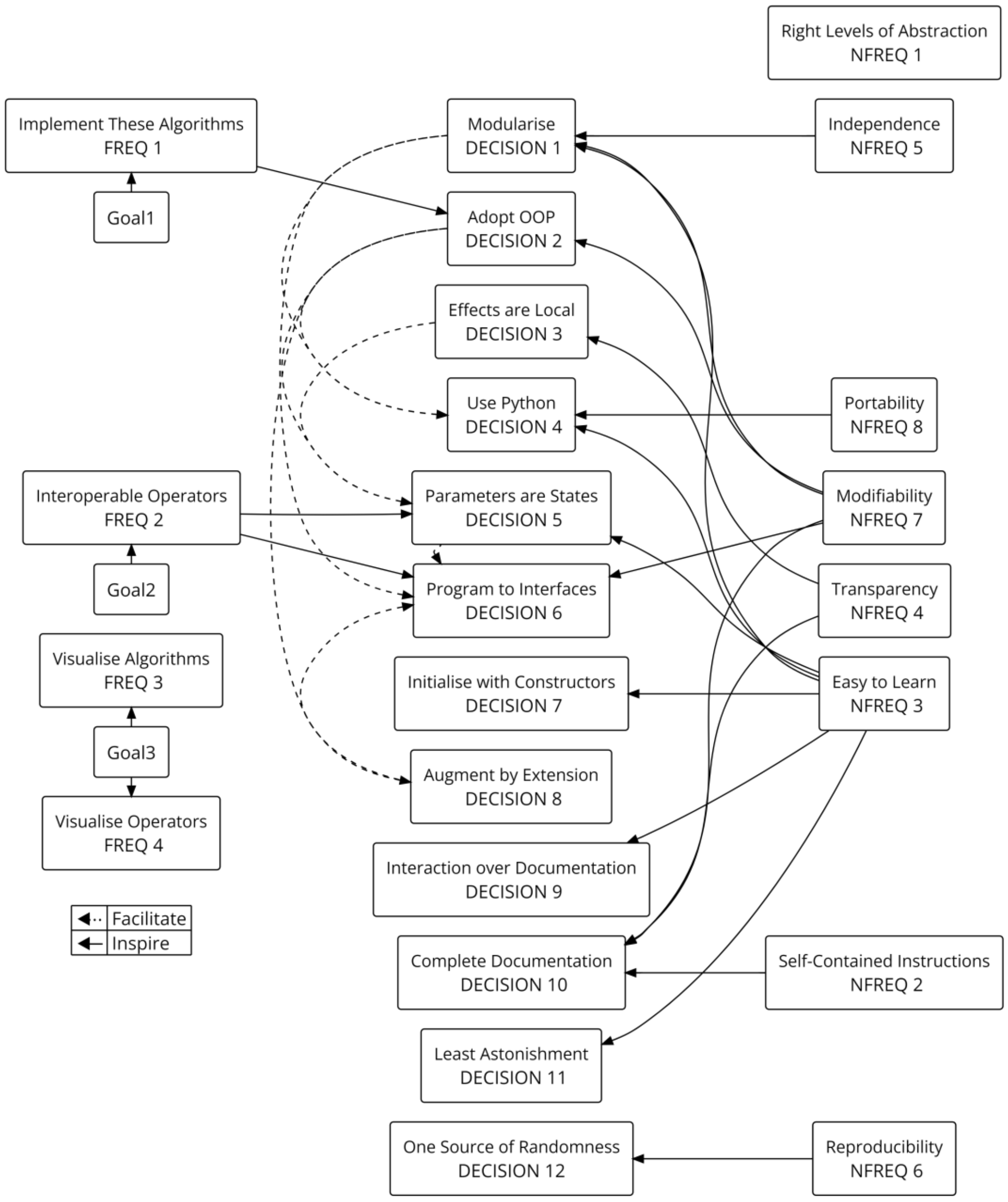
Right Levels of Abstraction NFREQ 1
Implement These Algorithms FREQ 1
Modularise DECISION 1
Independence NFREQ 5
Goal1
Adopt OOP DECISION 2
Effects are Local DECISION 3
Use Python DECISION 4
Portability NFREQ 8
Interoperable Operators FREQ 2
Parameters are States DECISION 5
Modifiability NFREQ 7
Goal2
Program to Interfaces DECISION 6
Transparency NFREQ 4
Visualise Algorithms FREQ 3
Initialise with Constructors DECISION 7
Easy to Learn NFREQ 3
Goal3
Augment by Extension DECISION 8
Visualise Operators FREQ 4
Interaction over Documentation DECISION 9
Facilitate
Inspire
Complete Documentation DECISION 10
Self-Contained Instructions NFREQ 2
Least Astonishment DECISION 11
One Source of Randomness DECISION 12
Reproducibility NFREQ 6

*Figure 5: Traceability diagram: goals to requirements to design decisions*

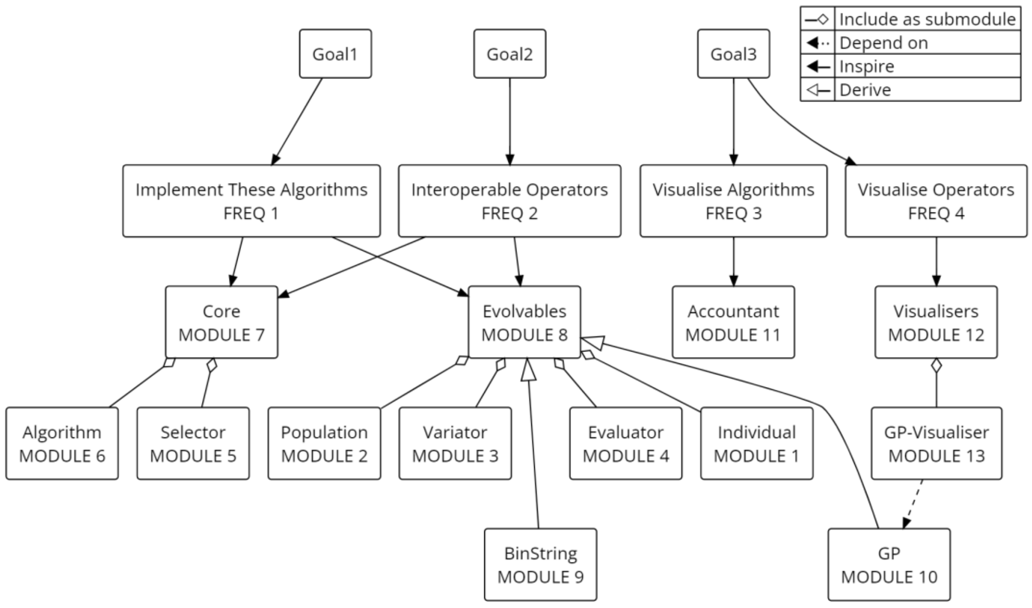

*Figure 6: Traceability diagrams: goals to requirements to modules*

## 5.1 Design Decisions

This section enumerates design decisions of EvoKit. Each decision traces back to a functional or nonfunctional requirement.

### 5.1.1 Modularise

module

«Modularise» DECISION 1: Modularisation decomposes the software into disjoint **modules**, so that each module is assigned a unique set of tasks, or *responsibility* [20, p. 1054]. A module can use other modules.

A modular design has many benefits:

- The assignment of responsibility to modules naturally groups relevant behaviours together. This helps with documentation and understanding of the project («Ease to Learn» NFREQ 3).
- Use relations between modules makes dependency relations explicit. In doing so, usage of external modules also becomes easy to check («Independence» NFREQ 5).
- The independent nature of modules means work done in one module should not affect another module. In addition, a module can be swapped out as long as the replacement has the same capabilities. This design enables modifications on a local scale («Modifiability» NFREQ 7).

### 5.1.2 Adopt OOP

object

member

«Adopt OOP» DECISION 2: The object-oriented programming (OOP) paradigm uses **objects**, entities that have states (*fields*) and behaviours (*methods*) known collectively as **members** [21, p. 9]. The nature of evolutionary algorithms lends to them being implemented as objects:

- An algorithm uses operators; an operator has parameters; the algorithm may inspect and change these parameters during runtime. This structure can be modelled with objects and members («Implement Algorithms» FREQ 1).
- Operators of the same type share common behaviours; for example, all evaluators map from an individual to a real vector. Representing operators as objects allows the user to change the behaviour of an operator by extension, without directly modifying the operator («Modifiability» NFREQ 7).

### 5.1.3 Effects are Local

«Effects are Local» DECISION 3: Unbound functions do not produce side effects; methods only affect states of the owner object or its arguments; the documentation explicitly mentions all effects.

effect

An **effect**, or side effect, is the consequence of invoking a function other than the returned value [22, p. 10]. Examples of side effects include changing a global variable, emitting to a file, or printing to the user interface.

Effects have the potential to change any part of the program. Similarly, the use of global states let a function take input from any part of the program. These two make it difficult to reason about the program. Limiting and clarifying effects improves transparency («Transparency» NFREQ 4).

### 5.1.4 Use Python

«Use Python» DECISION 4: Python is a popular programming language in the machine learning research community. This software is coded in Python for the following benefits:

- Python interpreters such as CPython are available in many operating systems and platforms, from microcontrollers [23] to major operating systems including Linux/UNIX systems, macOS, and Windows [24] («Portability» NFREQ 8).
- Python supports modules («Modularise» DECISION 1).
- Python supports objects («Adopt OOP» DECISION 2).
- Knowledge of Python is common in machine learning researchers. Documentation tools for Python (e.g. Sphinx) are widely used and actively supported by the community («Ease to Learn» NFREQ 3).

There are downsides to using Python. Empirical study showed that Python is significant slower to C, C++, and Java, though it is also faster to develop [25]. The ease to develop and use Python programs aligns with EvoKit's priorities.

### 5.1.5 Make Hyperparameters States

«Parameters are States» DECISION 5: Assign hyperparameters to an operator as attributes, not when methods of the operator are called. That is, associate states with its owner, not its user. This is only possible in a design that uses objects («Adopt OOP» DECISION 2).

This approach localises effects on parameters to just the parent object («Effects are Local» DECISION 3). Grouping parameters with most relevant objects also promotes understandability («Ease to Learn» NFREQ 3).

elitist selector

To elaborate, consider an **elitist selector** $s$ with parameter best_individual. This parameter stores the best individual encountered so far, so that a duplicate of it is always deposited into the result of selection. This parameter could be stored in $s$ as an attribute (see Algorithm 1), or it could be managed by the algorithm as a local variable (see Algorithm 2):

Algorithm 1: Storing hyperparameter as attribute

```
$s \in S \leftarrow \textit{initialise\_selector}()$
$s$.best_individual $\in \mathcal{I} \leftarrow$ old_population.best_individual
new_population $\in \mathcal{I} \leftarrow s.\textit{select}$(old_population)
```

Algorithm 2: Storing hyperparameter as local variable

```
$s \in S \leftarrow \textit{initialise\_selector}()$
best_individuall $\in \mathcal{I} \leftarrow$ old_population.best_individual
new_population $\in \mathcal{I} \leftarrow S.\textit{select}$ (old_population, best_individual)
```

In Algorithm 1, the operator stores its parameters; in Algorithm 2, the algorithm maintains parameters of all operators.

### 5.1.6 Program to Interface

«Program to Interface» DECISION 6: Design interfaces before implementation; program against interfaces. Interfaces are part of modules («Modularise» DECISION 1).

interface

secret

An **interface** only describes how to use a module [26]. As such, the interface hides all **secret**s – details that underly the module's behaviour. This separation of concern allows secrets to change freely, as long as these changes do not affect how the module can be used («Modifiability» NFREQ 7) [27].

At the same time, modules that achieve the same purpose in different ways can share the same interface. This is useful for designing interoperating operators («Interoperable Operators» FREQ 2).

Because parameters are stored as states («Parameters are States» DECISION 5), methods do not have to accept parameters in arguments. This allows operators of the same type to share a common set of methods.

### 5.1.7 Initialise with Constructors

«Initialise with Constructors» DECISION 7: Algorithms and operators receive parameters in constructors.

At minimum, an evolutionary algorithm must accept operators that dictate its behaviour and each operator can accept more parameters. This can be done in two ways: via setters or via constructors.

For example, DEAP takes the setter approach. An algorithm in this framework (Example 1) begins with the initialisation of an empty `Toolbox`, followed by a sequence of calls to `Toolbox.register`. The tutorials instruct which attributes need to be set[1].

[1] http://deap.readthedocs.io/en/master/overview.html

Example 1: Workflow of creating an algorithm in DEAP

```
  ⋮
toolbox = base.Toolbox()
# Attribute generator
toolbox.register("attr_bool", ...)
# Structure initializers
toolbox.register("individual", ...)
toolbox.register("population", ...)
  ⋮
toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", ...)
  ⋮
```

The constructor approach lists all necessary parameters in constructors. In pymoo for example (Example 2), the constructor of `NSGA2` (representing algorithm NSGA2) receives parameters such as pop_size, n_offsprings, and mutation[2]. Doing so reduces the need to consult the documentation («Ease to Learn» NFREQ 3).

[2] https://pymoo.org/getting_started/part_2.html

Example 2: Workflow of creating an algorithm in pymoo

```
from ... import NSGA2
from ... import SBX
from ... import PM
from ... import FloatRandomSampling

algorithm = NSGA2(
    pop_size=40,
    n_offsprings=10,
    sampling=FloatRandomSampling(),
    crossover=SBX(prob=0.9, eta=15),
    mutation=PM(eta=20),
    eliminate_duplicates=True
)
```

EvoKit takes a hybrid approach: an object in this framework accepts essential parameters in the constructor and accepts non-essential parameters via setters. Figure 7 provides a visual illustration.

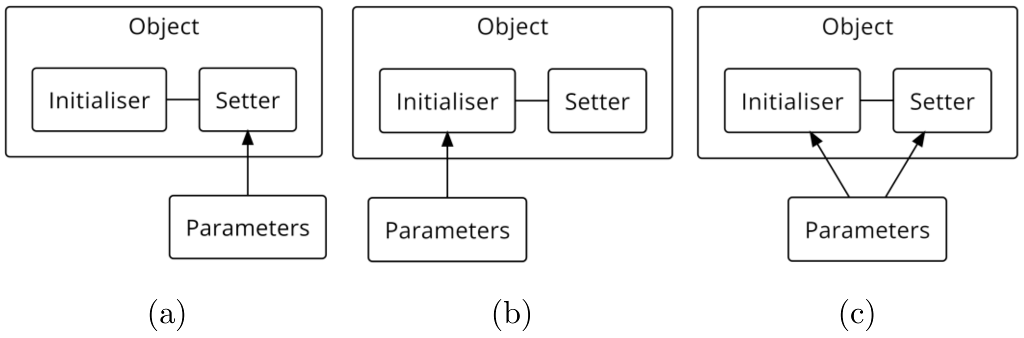

*Figure 7: Passing parameters by (a) setter, (b) constructor, and (c) both.*

Also note that in pymoo, parameters such as the mutation rate and number of offsprings are given to the algorithm itself. EvoKit attaches these parameters to most relevant operators – mutation rate to variators and number of offsprings to selectors («Parameters are States» DECISION 5.)

### 5.1.8 Modify by Extension

«4.1.8 Modify by Extension» DECISION 8: The object-oriented design («Adopt OOP» DECISION 2) makes it possible to modify a class by extension. This pattern is also used in popular frameworks such as PyTorch and Keras [28] [29].

extension **Extending** (or deriving) a class changes its implementation («Modifiability» NFREQ 7) without changing existing elements of its interface («Program to Interface» DECISION 6). For example, all neural networks in PyTorch extend class `nn.Module` and, in doing so, inherit the `forward` method. Consequently, all neural networks can be trained in the same way by calling this method.

Example 3: Definition of a neural network in PyTorch

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = ...
        self.linear_relu_stack = ...

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear_relu_stack(x)
        return x
```

### 5.1.9 Interaction over Documentation

«Interaction over Documentation» DECISION 9: For each major feature, the framework provides an interactive tutorial.

This requirement is sourced from Keras [30]. The framework should allow the user to learn from examples. The documentation should act as a fallback, for advanced users and dealing with situations that are not explained in examples («Ease to Learn» NFREQ 3).

The interactive tutorials are located here.

### 5.1.10 Complete Documentation

«Complete Documentation» DECISION 10: Every public interface and every private interface is documented. The documentation describes every effect, every special behaviour, and every exception that can be raised.

documentation

**Documentation** is vital to using and maintaining code. The documentation is the final authority on how the code operates, which is vital to both learning and explaining the framework («Self-Sufficient Instructions» NFREQ 2, «Transparency» NFREQ 4).

In addition, the documentation captures valuable insights into development, which enables maintenance and updating the software for future needs («Modifiability» NFREQ 7).

The documentation is located here.

### 5.1.11 Least Astonishment

«Least Astonishment» DECISION 11: Introduce new concepts slowly and only when necessary. Only use concepts from evolutionary computing or software engineering.

The Keras API design guidelines mention the need to reduce cognitive load [31]. In short: the framework should introduce as few new concepts as possible. When new concepts are introduced, they should be taken from either the problem domain (evolutionary computing) or the solution domain (software engineering). Only then can the framework be easily learned and used («Ease to Learn» NFREQ 3).

### 5.1.12 One Source of Randomness

«One Source of Randomness» DECISION 12: Use the `random` module as the sole source of randomness. Having only one source of randomness improves reproducibility («Reproducibility» NFREQ 6).

Custom frameworks can implement their own random seeds, such as `torch.manual_seed` for PyTorch and `np.random.seed` for NumPy [32]. the framework instead uses the native `random.seed` method.

Note that a user-defined operator may use its own source of randomness. For example, a evaluator that runs simulations can source randomness from an external module. The framework cannot and should not prevent this.

## 5.2 Module Design

Following «Modularise» DECISION 1, the framework divides into modules. This section discusses the design of module interfaces and how modules combine to form a workflow.

### 5.2.1 Global and Local Strategies

In an evolutionary algorithm, all operators of the same type share the same responsibility (see Table 3). This feature presents an opportunity to design one interface for each operator type («Program to Interface» DECISION 6).

The behaviour of an operator is defined on two levels – as a *global strategy* and a *local strategy*. Take three implementations of selectors for example, where let $n$ is the target number of offspring:

truncation selection

◊ **Truncation selection** [33, p. 372] (or simply top-$n$ selection [34, p. 89]) takes $n$ highest-fitness individuals from the population.

tournament selection

- ◊ **Tournament selection** [33, p. 368] [34, p. 86] takes the best individual from a uniform random subset of the population; repeating $n$ times.

fitness proportionate selection

- ◊ **Fitness proportionate selection** [34, p. 81] selects an individual with probability proportional to its fitness; repeating $n$ times.

These selectors are defined in the same general format: from a *subset of the population* (determined by the global strategy), select a *subset* (using the local strategy). Table 5 defines the above selectors in this format:

TABLE 5: OPERATORS, BY GLOBAL AND LOCAL LEVEL STRATEGIES

| **Selector** | **Global Strategy** | **Local Strategy** |
|---|---|---|
| Truncation | From all | Select best $n$ |
| Tournament | From subset | Select best $n$ |
| Fitness proportionate | From all | Select each with probability proportional to fitness. |

Therefore, EvoKit implements each operator in two parts (Algorithm 3): the global behaviour and the local behaviour, where the global behaviour acts a context for the local behaviour. When expending an operator, the user can choose to override neither, either, or both behaviours.

Algorithm 3: Behaviour of the Selector module

```
class Selector
    function select_from_population ($P \in \mathcal{P}$)$\rightarrow \mathcal{P}$
        $P^* \in \mathcal{P} \leftarrow$ *initialise_empty_population* ()
        repeat until *is_empty* ($p$) or *a termination condition is met*
            $P' \in \mathcal{P} \leftarrow$ *global_selection_strategy* ($P$)
            $P'^* \in \mathcal{P} \leftarrow$ select_from_subset($P'$)
            update $P \leftarrow P \setminus P'^*$
            $P^* \leftarrow P^* \cup P'^*$
        end repeat
        return $P^*$
    end function
    function select_from_subset($P' \in \mathcal{P}$)$\rightarrow \mathcal{P}$
        $P'^* \in \mathcal{P} \leftarrow$ *local_selection_straetgy* ($P'$)
        return $P'^*$
    end function
end class
```

This design has another benefit: to modify an operator by extension, the user can override just the global or local selection strategy («Modifiability» NFREQ 7).

### Variators

The design of variators is similar (Algorithm 4). Here, for each local strategy that receives $n$ parents, the default global strategy splits the population into $n$-tuples then supplies each tuple to the local strategy. The global strategy then collects the results then returns them.

Algorithm 4: Behaviour of the Variator module

```
class Variator
    function initialise (arity $\in \mathbb{Z}_{0+}$)
    ⋮
    function vary_population ($P \in \mathcal{P}$)$\rightarrow \mathcal{P}$
        $P^* \in \mathcal{P} \leftarrow$ initialise_empty_population ()
        until $p$ is exaused
            $P' \in \mathcal{P} \leftarrow$ next $n$ items in $\mathcal{P}$
            $P^* \in \mathcal{P} \leftarrow P^* \cup$ vary($P'$)
        end until
        return $P^*$
    end function
    function vary($P' \in \mathcal{P}$)$\rightarrow \mathcal{P}$
        $P'^* \in \mathcal{P} \leftarrow$ local_variation_strategy ($P'$)
        return $P'^*$
    end function
end class
```

**Evaluators**

For evaluators, the default global strategy applies a local strategy to each item in the population.

Algorithm 5: Behaviour of the Evaluator module

```
class Evaluator
    function evaluate_population ($P \in \mathcal{P}$)$\rightarrow$ effect
        for each $r$ in $P$
            update $r$.fitness: $\mathbb{R} \leftarrow$ evaluate($r$)
        end for
    end function
    function evaluate($r \in \mathcal{I}$)$\rightarrow \mathbb{R}$
        return local_evaluation_strategy($r$)
    end function
end class
```

### 5.2.2 Chaining Operators

The evaluator, selector, and variator map from and to just three sets: $\mathcal{P}$, $\mathcal{P}_E$, and $\mathcal{P}_U$ (see Table 3). The following operators or compositions of operators map from $\mathcal{P}$ to subsets of $\mathcal{P}$. They can therefore be chained.

TABLE 6: TYPES OF OPERATORS OR COMPOSITIONS OF OPERATORS

| **Operator or Composition** | **Type** |
|---|---|
| evaluator $\circ$ selector | $\mathcal{P} \to \mathcal{P}$ |
| variator | $\mathcal{P} \to \mathcal{P}_U, \qquad \mathcal{P}_U \subset \mathcal{P}$ |

Consequently, the framework treats an iteration as a composition of two operations: (a) evaluation followed by selection and (b) variation. This formulation has two benefits:

### Ease to Write Algorithms

Because both operations map from the same set to its subset, one can write an algorithm by constantly reassigning to the same variable, updating the population at each step.

Algorithm 6: One iteration of a simple evolutionary algorithm

```
function step($P \in \mathcal{P}$,  $v \in V$,  $e \in E$,  $s \in S$)$\rightarrow$ *effect*
    update $P \in \mathcal{P}_U \leftarrow v(P)$  *apply the variator*
    update $P \in \mathcal{P}_E \leftarrow e(E)$  *apply the evaluator*
    update $P \in \mathcal{P}_E \leftarrow s(P)$  *apply the selector*
end function
```

**Ease to Visualise Algorithms**

An algorithm written in this form can be visualised as an acyclic graph, where nodes are operators and edges represent the passing of individuals between operators. For example, Figure 8 visualises Algorithm 6 as a sequence of assignments to the same variable:

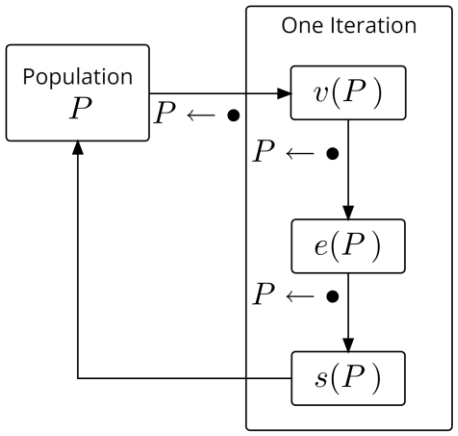

*Figure 8: Algorithm 6 as a sequence of assignments*

This formulation also significantly increases the range of algorithm that EvoKit can implement. For example, Figure 9 visualises a steady-state algorithm, where each iteration replaces only a subset of the population [35, p. 510].

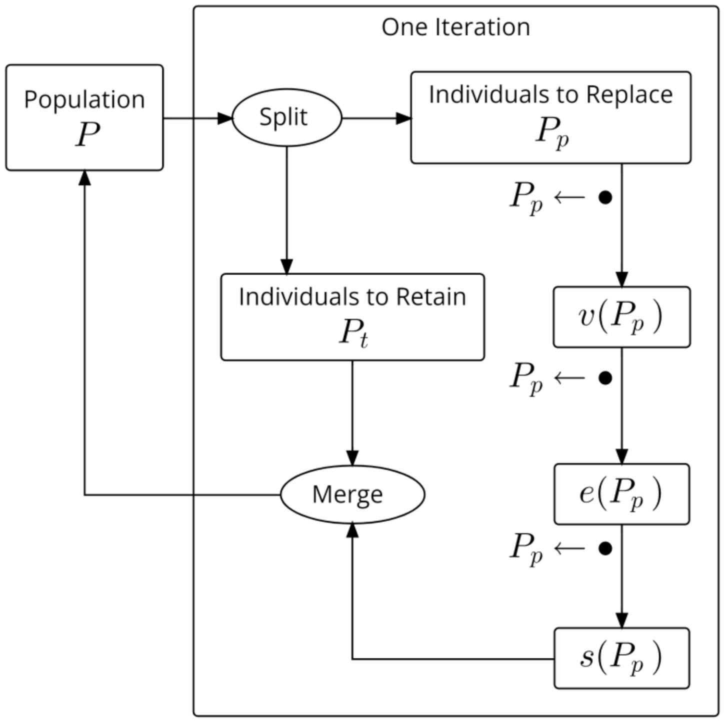

*Figure 9: Visualisation of a steady-state algorithm as an acyclic graph*

## 5.3 Modules

abstract base class

The framework implements one module for each component of an evolutionary algorithm, in addition to the algorithm itself. These modules are **abstract base class**es that specify the behaviours of concrete modules of that type.

### 5.3.1 Individual

«Individual» MODULE 1: Generic base class of all individuals. The individual includes the representation, as well as parameters that relate to the training process (e.g. fitness).

Offspring produced by the variator must be independent from parents, so that changes made on either the offspring of its parent does not affect the other. To enable this, Individual must implement a method that creates an independent copy of itself.

Table 7: Members of an individual

| **Module** | **Member** | **Type** | |
|---|---|---|---|
| Individual | genome | Depends on the representation | |
| | fitness | Tuple[float] | (Tuple of float) |
| | copy | Self | (Same class) |

### 5.3.2 Population

«Population» MODULE 2: The population represents a sequence of individuals.

The population is generic over type parameter D, covariant to Individual.

Table 8: Members of a population

| **Module** | **Type** | |
|---|---|---|
| Population[D] | Sequence[D] | (Generic sequence) |

### 5.3.3 Variator

«Variator» MODULE 3: This module specifies the common interface of variators. All variator implementations must use this interface.

The variator is generic over type parameter D, covariant to Individual.

TABLE 9: MEMBERS OF A VARIATOR

| Module | Member | Type |
|---|---|---|
| Variator[D] | vary | self $\times$ Population[D] $\rightarrow$ Population[D] |
| | vary_population | self $\times$ Population[D] $\rightarrow$ Population[D] |

### 5.3.4 Evaluator

«Evaluator» MODULE 4: This module specifies the common interface of evaluators. All evaluator implementations must use this interface.

The evaluator is generic over type parameter D, covariant to Individual.

TABLE 10: MEMBERS OF AN EVALUATOR

| Module | Member | Type |
|---|---|---|
| Evaluator[D] | eval | self $\times$ D $\rightarrow$ float |
| | eval_population | self $\times$ D $\rightarrow$ *effect* |

### 5.3.5 Selector

«Selector» MODULE 5: This module specifies the common interface of selectors. All selector implementations must use this interface.

| **Module** | **Member** | **Type** |
|---|---|---|
| Selector | budget | int |
| | select | self × Population[D] → Sequence[D] |
| | select_population | self × Population[D] → Population[D] |

### 5.3.6 Algorithm

«Algorithm» MODULE 6: This module specifies the common interface of evolutionary algorithms. All concrete evolutionary algorithms must use this interface.

Invoking the .step method advances the population by one generation.

| **Module** | **Member** | **Type** |
|---|---|---|
| Algorithm | step | Nothing → *effect* |

### 5.3.7 Grouping Modules

Machine learning algorithm receives an input, performs some computation, then emits an output [2, p. 3]. This division of responsibilities leads to a grouping of modules as shown in Figure 10. These modules satisfy requirements «Implement Algorithms» FREQ 1, «Interoperable Operators» FREQ 2, and «Right Level of Abstraction» NFREQ 1.

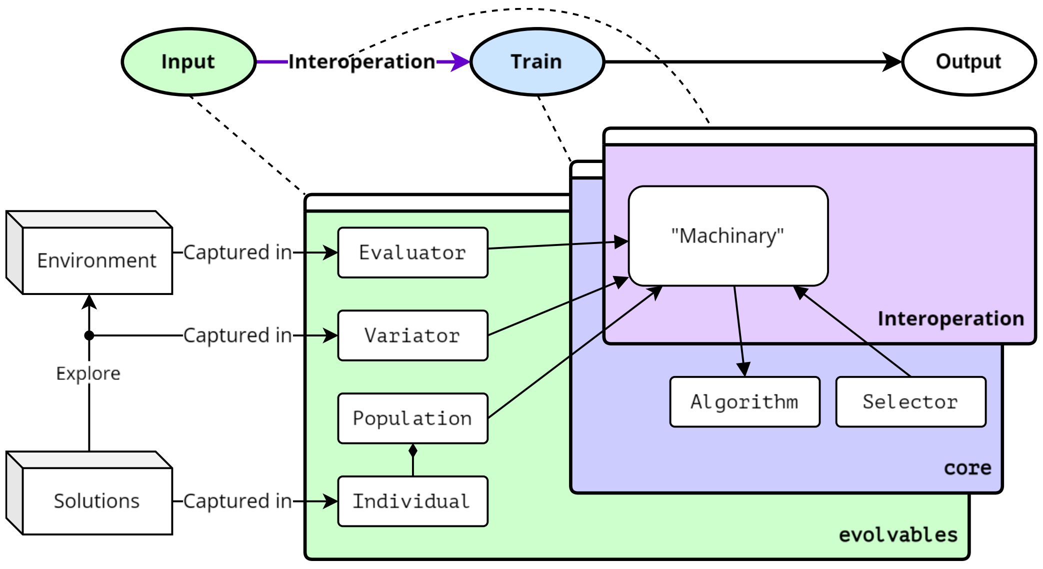

*Figure 10: Division of modules by responsibility*

The following modules contain submodules that represent algorithm components, from «Individual» MODULE 1 to «Algorithm» MODULE 6.

«Core» MODULE 7: The Core module manages the learning process. Its submodules are agnostic to the problem: the algorithm and the selector can be used with any combination of representations, evaluators, and variators.

«Evolvables» MODULE 8: The Evolvables module includes submodules that directly interact with the problem, namely: (a) the individual that captures a solution, (b) the evaluator which captures how well a solution solves the problem, and (c) the variator which explores the solution space by deriving new solutions from existing ones.

### 5.3.8 Implementation Modules

The framework ships with two implementations. Each implementation captures a well-known problem in evolutionary computing and a representation that has been widely used to solve it.

«BinString» MODULE 9: Capture the OneMax problem and genetic algorithms, using binary string representations.

«GP» MODULE 10: Capture the symbolic repression problem and genetic programming, using expression tree representations.

### 5.3.9 Accountant

Because an evolutionary algorithm is iterative, it is trivial to access the state of a population between iterations. However, what happens during an iteration is opaque to the user; the framework should provide a way to visualise this «Visualise Algorithms» FREQ 3.

Because an algorithm is an acyclic graph of operators, there are two places where useful information can be generated: inside operators and between operators. Because operators are defined by the user, the framework may not be able to see into operators. The framework can still report what happens between operators, however. Figure 11 illustrates this.

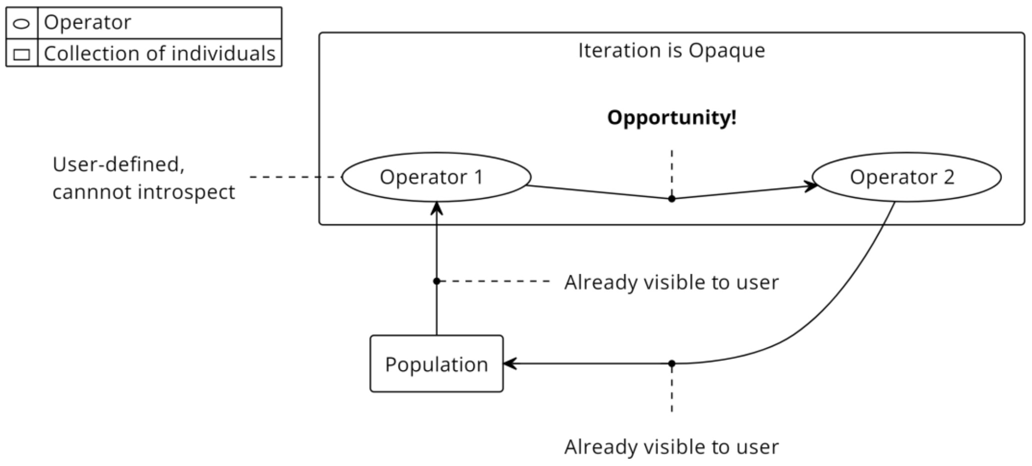

*Figure 11: Opportunity to implement a statistics module*

observer pattern

«Accountant» MODULE 11: The solution to this is the **observer pattern** [21, pp. 336-351]. As shown in Figure 12, the following exchanges occur between the Algorithm and Accountant modules:

1. The algorithm registers an accountant.
2. Then, the algorithm subscribes the accountant to itself.
3. When an event fires in the algorithm, the algorithm notifies all attached accountants.
4. When the accountant receives a notification, it collects data from the associated algorithm.

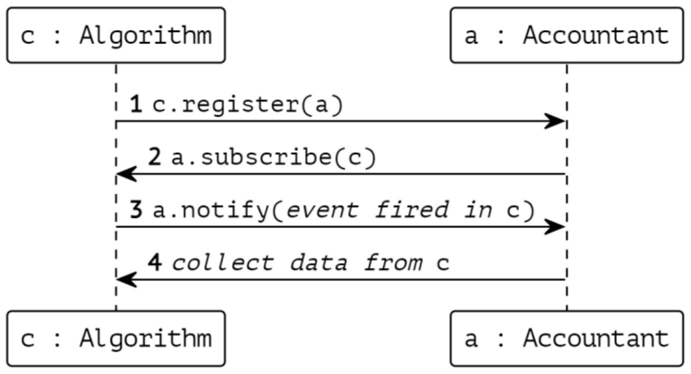

*Figure 12: Information exchanged between Algorithm and Accountant*

### 5.3.10 Visualisers

Some complex representations, such as tree-based genetic programs [36], are difficult to represent with text. The Visualiser module represents these individuals as figures («Visualise Individuals» FREQ 4). Its submodules correspond to representations.

«Visualisers» MODULE 12: Visualiser for complex representations.

«GP-Visualiser» MODULE 13, submodule of «Visualisers» MODULE 12: Visualiser for tree-based genetic programs.

## 5.4 The Workflow: Bring Everything Together

The module SimpleLinearAlgorithm extends Algorithm. SimpleLinearAlgorithm describes a simple three-step algorithm of variation-selection-evaluation., as shown in Figure 13.

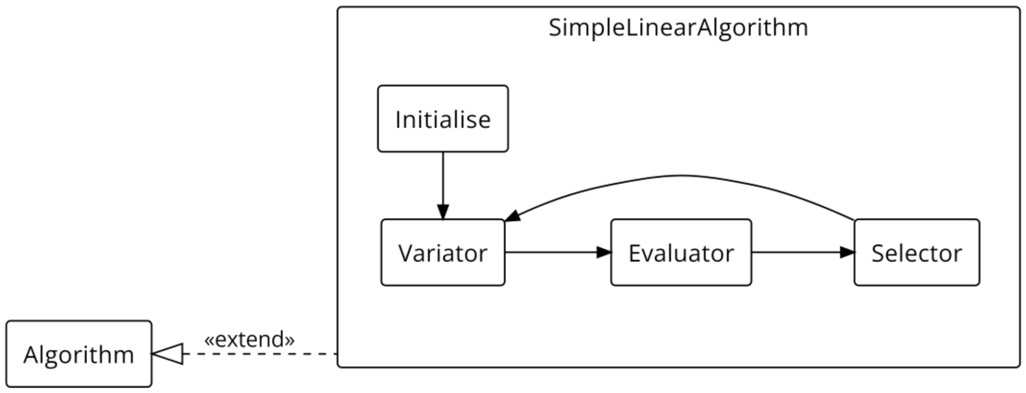

*Figure 13: A simple linear algorithm*

With the algorithm defined, the user can define custom operators. Each custom operator should extend the corresponding base class (Evaluator, Selector, and Variator). Figure 14 illustrates these operators and their inheritance relations.

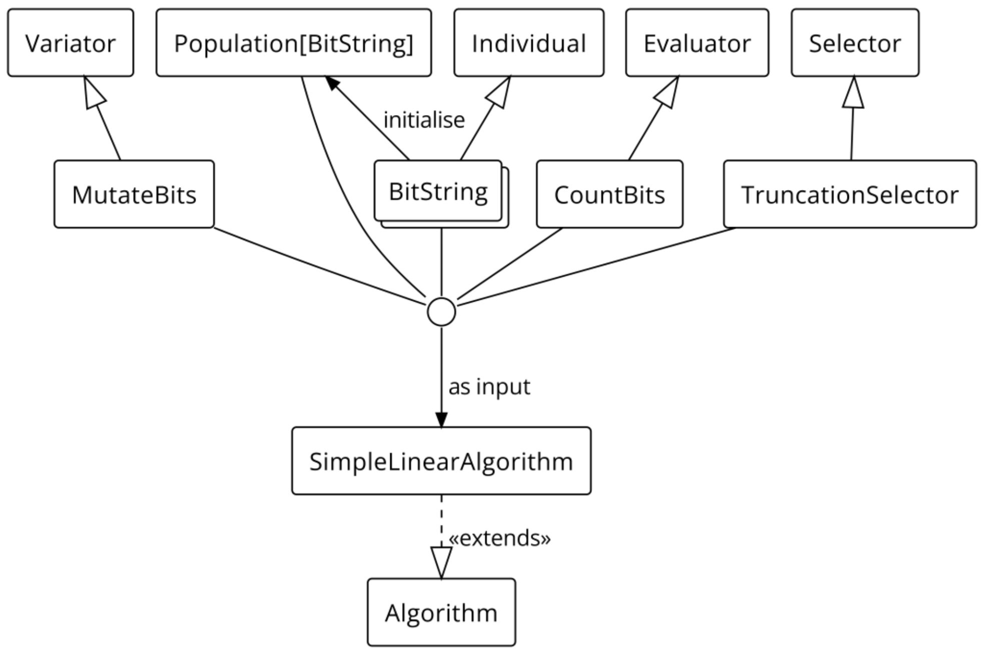

*Figure 14: Initialising an algorithm with components*

While all of that seems intimidating, it can be done in just 5 lines: 1 line to define the algorithm, 1 line to initialise the population, and 3 lines to initialise operators. Example 4 shows an example of this.

Example 4: Invoking an evolutionary algorithm in 5 lines

```
ctrl: SimpleLinearAlgorithm = SimpleLinearAlgorithm(
  population=Population(*[BinaryString.random() \
    for _ in range(POPULATION_SIZE)]),
  variator=MutateBits(mutation_rate=0.1),
  evaluator=CountBits(),
  selector=TruncationSelector[BinaryString],)
```

## 6 Examples

This section demonstrates the application of the framework to solve two problems: OneMax and Symbolic Regression.

### 6.1 OneMax

This section demonstrates the application of genetic programming to solve a symbolic regression problem. The algorithm listed in this section uses the following components:

Table 11: Choices for each component type for OneMax

| Type | Choice | Definition |
|---|---|---|
| ***Algorithm*** | Minimal | Figure 2 |
| ***Representation*** | Binary string | 6.1.1 Components |
| ***Evaluator*** | OneMax | |
| ***Variator*** | Bit Mutation | 6.1.1 Components |
| ***Selector*** | $(\mu + \lambda)$ | 6.1.1 Components |

### 6.1.1 Components

OneMax

The "hello world" of evolutionary algorithms, the **OneMax** problem was first proposed for study with genetic algorithms [37].

#### Representation and Evaluator

The OneMax problem uses fixed-length binary string representations. It seeks to evolve a string whose bits sum to the largest possible number.

| | | |
|---|---|---|
| ***Representation*** | Fixed-length binary string | e.g. [1,1,0,1] |
| ***Evaluator*** | Sum digits | e.g. return 3 for [1,1,0,1 |

### Variator

bitwise mutation operator

Many variators have been defined for this representation. This example uses the **bitwise mutation operator** [2, p. 52]. For each bit in the representation, this operator flips that bit with probability $p \in [0 \ldots 1] \subseteq \mathbb{R}$.

| | |
|---|---|
| ***Variator*** | For each bit in the parent, flip it with probability $p$. |

### Selector

$(\mu + \lambda)$ selection

This example uses $(\mu + \lambda)$ **selection**, where $\mu$ parents produce $\lambda$ offspring. Then, parents and offspring form a pool of size $(\mu + \lambda)$, from which $\mu$ individuals with highest fitness are selected [2, p. 89].

| | |
|---|---|
| $(\mu + \lambda)$ ***Selector*** | Truncation selector, select $\lambda$ highest-fitness individuals from a combined pool of parents and offspring. |

### Example for One Iteration

This example uses 3 parents. Since the binary mutation operator produces one offspring for each parent, the number of offspring is also 3. The selector selects 3 individuals from these 6 $(3 + 3)$ individuals. Each individual has two memberss: the genome (the binary string representation) and its fitness.

Table 12: Individuals in the example algorithm

| **Individual** | **Members** | |
|---|---|---|
| | genome | fitness |
| $p_1$ | [1,0,0,0,1] | sum(1,0,0,0,1) = 2 |
| $p_2$ | [0,1,1,1,0] | sum(0,1,1,1,0) = 3 |
| $p_3$ | [0,1,0,1,0] | sum(0,1,0,1,0) = 2 |

**Offspring Generation**

The variator applies to each of $p_1, p_2, p_3$. For each digit in each $p_i$, the variator independently decides whether to flip that digit, with probability $p$ to flip.

Suppose that the decisions are $[y, y, n, n, y]$, $[n, n, y, n, y]$, and $[n, y, y, y, y]$ for $p_1, p_2, p_3$ respectively. XORing a decision to the bit string applies that decision.

$$\begin{array}{llll} p_1 = [1,0,0,0,1] & \xrightarrow{XOR} & \overbrace{[1,1,0,0,1]}^{\text{decisions as masks}} & \rightarrow [0,1,0,0,0] = p'_1 \\ p_2 = [0,1,1,1,0] & \xrightarrow{XOR} & [0,0,1,0,1] & \rightarrow [0,1,0,1,1] = p'_2 \\ p_3 = [0,1,0,1,0] & \xrightarrow{XOR} & [0,1,1,1,1] & \rightarrow [0,0,1,0,1] = p'_3 \end{array}$$

**Selection**

Together, parents and their offspring form a pool of size 6.

TABLE 13: INDIVIDUALS IN THE EXAMPLE ALGORITHM, POST VARIATION

| Individual | Members | |
|---|---|---|
| | Genome | Fitness |
| $p_1$ | [1,0,0,0,1] | sum(1,0,0,0,1) = 2 |
| $p_2$ | [0,1,1,1,0] | sum(0,1,1,1,0) = 3 |
| $p_3$ | [0,1,0,1,0] | sum(0,1,0,1,0) = 2 |
| $p'_1$ | [0,1,0,0,0] | sum(0,1,0,0,0) = 1 |
| $p'_2$ | [0,1,0,1,1] | sum(0,1,0,1,1) = 3 |
| $p'_3$ | [0,0,1,0,1] | sum(0,0,1,0,1) = 2 |

From there, the selector takes 3 individuals with highest fitness. That is: $p_2$ with fitness 3, $p'_2$ with fitness 3, and $p'_3$ with fitness 2. Because $p_1, p_3, p'_3$ share the same fitness of 2, $p'_3$ is selected uniformly to break the tie.

### 6.1.2 Analysis

The training curve (Figure 15) is generally smooth. Convergence occurs at step 40. The fitness improvement rate begins quickly, then plateaus close to convergence.

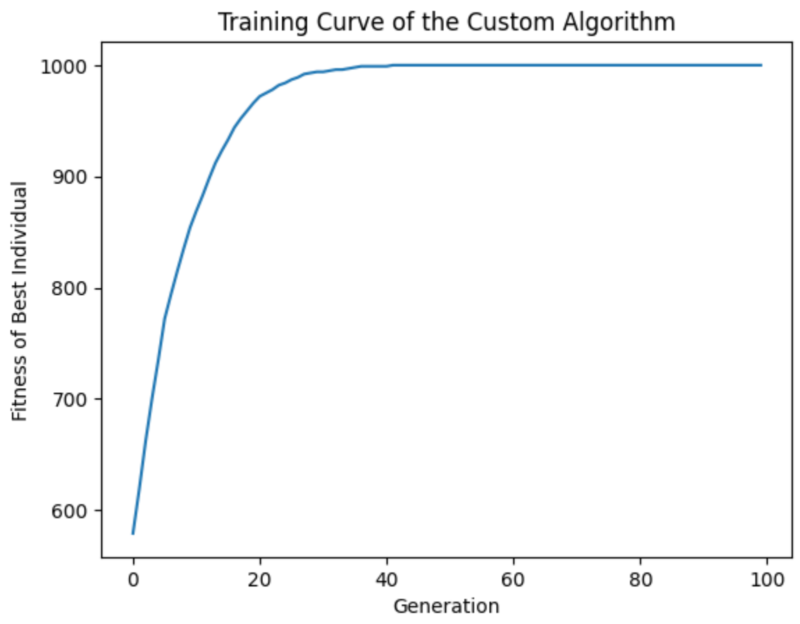

*Figure 15: Training curve of the custom algorithm*

## 6.2 Symbolic Regression

This section demonstrates the application of *genetic programming* to solve *symbolic regression*. The algorithm listed in this section uses the following components:

TABLE 14: HYPERPARAMETERS FOR SYMBOLIC REGRESSION

| **Type** | **Choice** | **Definition** |
|---|---|---|
| ***Algorithm*** | Minimal | Figure 2 |
| ***Representation*** | Expression tree | 6.2.2 Symbolic Regression |
| ***Evaluator*** | Symbolic regression | |
| ***Variator*** | Subtree crossover | 6.2.1 Generic Programming |
| ***Selector*** | $(\mu + \lambda)$ | 6.1.1 Components |

### 6.2.1 Generic Programming

generic programming

expression tree

**Generic programming** by Koza [36] uses representations that are themselves functions. Each representation (a **expression tree**, or **genetic program**) is a syntax tree where intermediate nodes are functions and terminal nodes are either nullary functions, constants, or variables.

As an example, the genetic program shown in Figure 16 is functionally identical to $\mathrm{pow}(\mathrm{add}(1, x_1), x_2)$, or $(1 + x_1)^{x_2}$.

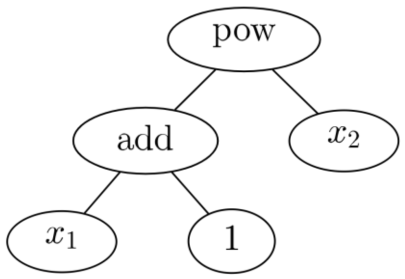

*Figure 16: Visualisation of a simple genetic program*

**Initialisation**

function set

terminal set

The initialisation of genetic trees uses several parameters. First, the **function set** and the **terminal set** decide values that could form nodes of the constructed program [2, p. 75]:

Table 15: A primitive set, sorted by arity

| **Set** | **Arity** | **Items** | **Definition** |
|---|---|---|---|
| Terminal | 0 | 2, 1, 0.5 | Real numbers |
| Function | 1 | sin | The sine function |
| | | cos | The cosine function |
| | 2 | add | $\mathrm{add}(x_1, x_2) \coloneqq x_1 + x_2$ |
| | | sub | $\mathrm{sub}(x_1, x_2) \coloneqq x_1 - x_2$ |
| | | mul | $\mathrm{mul}(x_1, x_2) \coloneqq x_1 \cdot x_2$ |
| | | div | $\mathrm{div}(x_1, x_2) = x_1/x_2$ if $x_2 \neq 0$, else 0 |

The construction of genetic tree induces a bias to the initial condition. This example uses a modified ramped half-and-half method [2, p. 105]. The process begins with a random root node, then recursively populates children of the node. Each random node contains a uniformly selected primitive of any arity, except that after a certain budget or depth is reached, only unary primitives (terminals) can be selected. Algorithm 7 describes the algorithm in more detail.

Algorithm 7: Initialisation of a random genetic tree

```
initialise global node_budget: int
initialise global MAX_DEPTH : int
new_tree : Tree ← make_random_tree(0)
function make_random_tree(current_depth: int) → Tree
    new_primitive: Primitive ← pool_primitive(current_depth)
    new_node: Node ← create_node_from(new_primitive)
    repat for arity(new_primitive) times
        new_node.add_child(make_random_tree(current_depth+1))
    end repeat
    return new_node
end function
function pool_primitive(current_depth: int) → Primitive, effect
    new_primitive: Node
    acess global node_budget
    acess global MAX_DEPTH
    if node_budget < 1 or current$s$_depth < MAX_DEPTH then
        new_primitive ← uniformly selected unary primitive
    else
        update node_budget ← node_budget − 1
        new_primitive ← uniformly selected primitive
    end if
    return new_primitive
```

```
end function
```

**Variator**

subtree crossover operator

This example uses the **subtree crossover operator**. For each pair of parents $(p_1, p_2)$, this operator performs the following actions:

1. Uniformly, select intermediate nodes $n_1 \in p_1$ and $n_2 \in p_2$
2. Uniformly, select arbitrary child $n_{1c}$ of $p_1$ and arbitrary child $n_{2c}$ of $p_2$
3. Exchange $n_{1c}$ and $n_{2c}$.

Figure 17 shows an example of this:

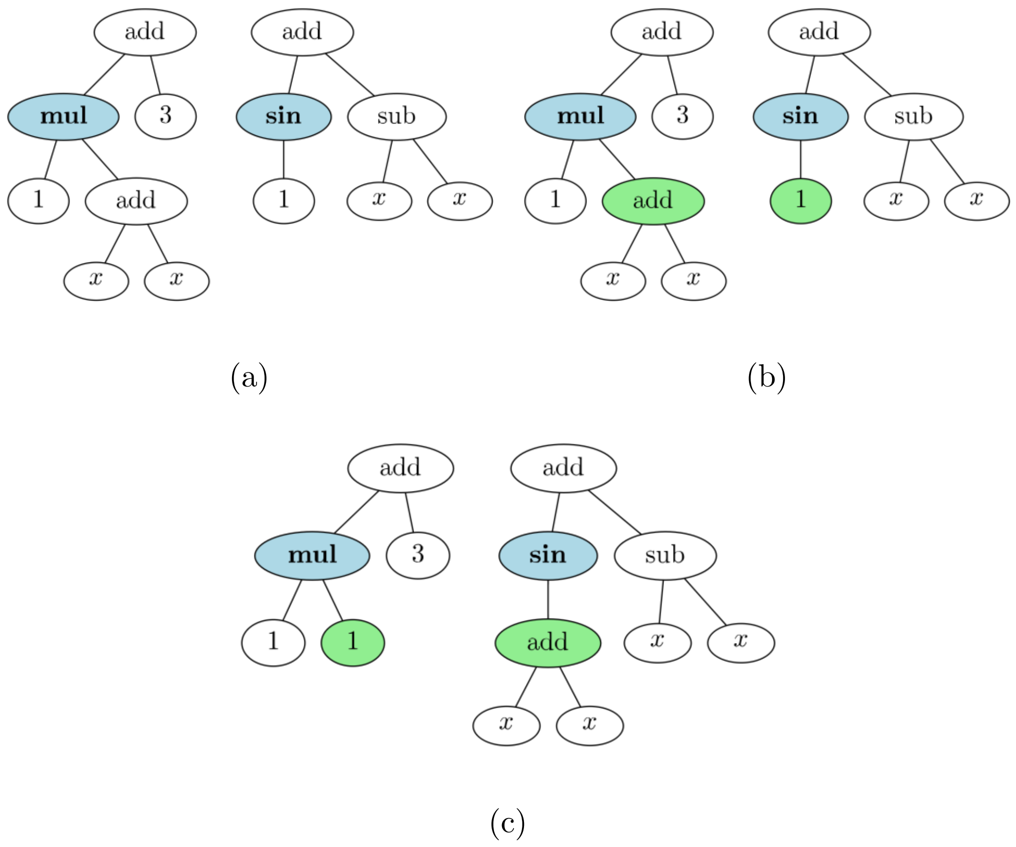

*Figure 17: Steps of a crossover operation: (a) select internal nodes, (b) select children of internal nodes, (c) result of crossover.*

### 6.2.2 Symbolic Regression

symbolic regression

The **symbolic regression** problem seeks to find a function that agrees most with a training set, or the underlying function that generated the training set [36].

**Evaluator**

Given a training set of points $S = \{x_i, y_i\}_{i=1}^{n}$ and a hypothetical generator $f^*$ where $y_i = f^*(x_i)$, the fitness of genetic program is the difference in its output from that of $f^*$:

$$\varphi(\, g \mid \{x_i, y_i\}_{i=1}^{n}\,) = \sum_{i=1,\dots,n} \text{abs}\big(y_i - g(x_i)\big)$$

In this example, the set $S$ is generated with $f_u = \sin(x) + 2\cos(x)$ over $\{-20, -19.75, -19.5, \dots ,19.5, 19.75\}$, or $\{(1-81)/4\}_{i=1}^{101}$.

As an example of how the evaluator works, let $g'$ be a representation of $\sin(x) + \cos(x)$. The fitness of $g'$, given evaluator $\varphi$, is

$$\sum_{i=1}^{161} \text{abs}[(\sin(x_i) + \cos(x_i)) - (\sin(x_i) + 2\cos(x_i))]\,, \qquad x_i = (i-81)/4.$$

### 6.2.3 Analysis

Figure 18 shows the training curve. Convergence occurs at generation 15.

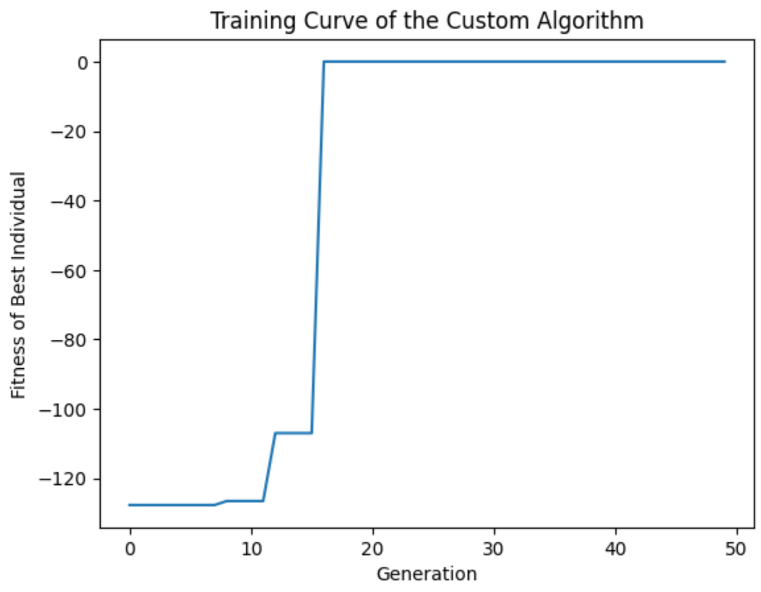

*Figure 18: Training curve of the symbolic regression algorithm*

Figure 19 plots highest-fitness individuals for each generation. Through generations, these individuals become better approximates of the test set, until convergence occurs.

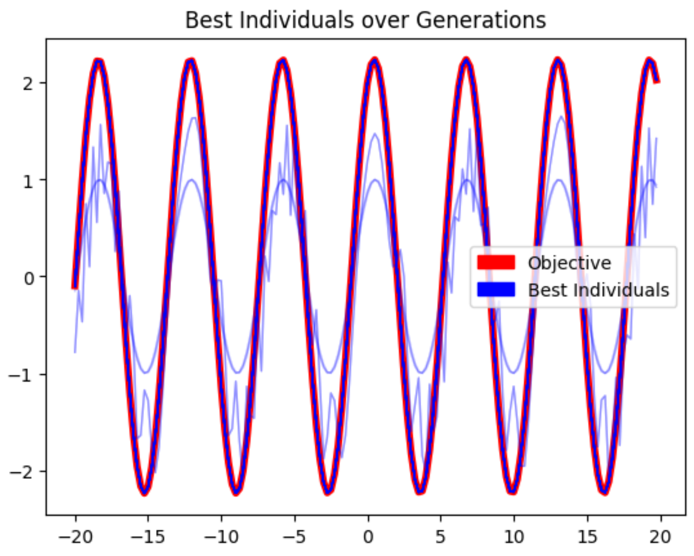

*Figure 19: Best individuals over generations*

Figure 20 shows the best evolved individual s$(\sin(x) + \cos(x) * 1) + \cos(x)$ which, after some simplification, becomes $\sin(x) + 2\cos(x)$.

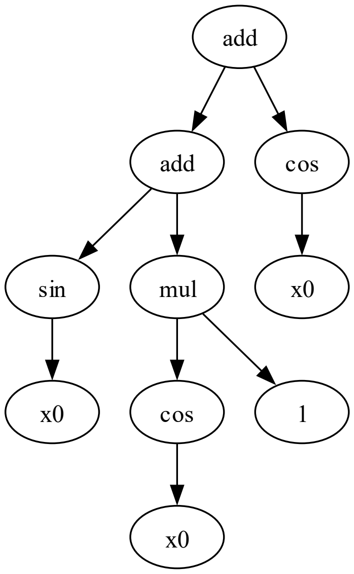

*Figure 20: Best evolved tree*

Note the occurrence of an identity function $* 1$. This is an instance of genetic programming – while the crossover operator does not seek to remove nodes, the algorithm is able to evolve extra nodes into identities, so that it can emulate a simple function with a representation that has more nodes.

# 7 Related Work

**EC-KiTy** is a well-engineered object-oriented evolutionary computing framework. Both EC-KiTy and EvoKit are object-oriented frameworks that explicitly evoke software engineering principles. I chose the name EvoKit in homage to EC-KiTy.

**PyTorch** is a state-of-the-art deep learning framework. EvoKit references PyTorch for its user-friendly design and usability features.

**Keras** is a high-level neural network API with a rich design guideline [31]. Its developers have written extensively about its design in blogs [30]. Several designs of EvoKit reference Keras.

**Pymoo** and **DEAP** are state-of-the-art evolutionary computing frameworks. They first inspired the development of EvoKit.

# 8 Future Work

During development, several compromises were made due to time constraint. Given time, the framework

**More algorithms:** The framework has only been used to implement genetic algorithms and genetic programming. It could implement other paradigms such as evolutionary programming, evolutionary strategies, multi-objective evolution, and more.

**Parallel computing**: Parallelism significantly speeds up evolutionary algorithms [38] [39]. Many frameworks [40] [41] [42] [43] implement parallel computing and even GPU computing. EvoKit must do the same.

**Linear Genetic Programming:** Many recent advancements [44] [45] in evolutionary computing are due to linear genetic programming, an alternative to tree-based genetic programs [46, p. 6]. This framework should implement linear genetic programming in the future.

## 9 Acknowledgements

To my supervisor Dr. Stephen Kelly for his unwavering support. Development of EvoKit began as part of an assignment in his class. His continued support has been vital in every stage of this project.

To Dr. Spencer Smith, who allowed me to bring this project into his class. His lessons on software engineering were instructed EvoKit's development from beginning to end.

Sections of this report are developed from a template by Smith and Lai [19].

# 10 Appendices

Appendix 1: Evolutionary Algorithms Implemented

| **Algorithm** | **Problem** | **Reference** |
|---|---|---|
| Genetic Algorithm | OneMax | [47] |
| Genetic Programming | Symbolic Regression | [36] |

Index

## 10.1 List of Tables

## 10.2 List of Figures

## 10.3 List of Algorithms

## 10.4 List of Functional Requirements

## 10.5 List of Nonfunctional Requirements

## 10.6 List of Design Decisions

## 10.7 List of Modules

## 11 References

[1] W. Banzhaf, P. Machado and M. Zhang, Handbook, 1.1 Introduction ed., Singapore: Springer Nature, 2024, pp. 4, 5.

[2] A. Eiben and J. Smith, Introduction To Evolutionary Computing, vol. 45, Springer, 2015.

[3] S. Dower, "Disambiguating evolutionary algorithms: composition and communication with ESDL.," 2012.

[4] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau and C. Gagné, "DEAP: Evolutionary Algorithms Made Easy," *Journal of Machine Learning Research,* vol. 13 , pp. 2171--2175, 7 2012.

[5] B. J. and D. K., "pymoo: Multi-Objective Optimization in Python," *IEEE Access,* vol. 8, pp. 89497-89509, 2020.

[6] M. Sipper, T. Halperin, I. Tzruia and A. Elyasaf, "EC-KitY: Evolutionary computation tool kit in {Python} with seamless machine learning integration," *SoftwareX,* vol. 22, p. 101381, 2023.

[7] M. Sipper, R. S. Olson and J. H. Moore, “Evolutionary computation: the next major transition of artificial intelligence?,” *BioData Mining,* vol. 10, no. 1, p. 26, 29 July 2017.

[8] J. D. Lohn, G. S. Hornby and D. S. Linden, “An Evolved Antenna for Deployment on Nasa’s Space Technology 5 Mission,” in *Genetic Programming Theory and Practice II*, U. O'Reilly, T. Yu, R. Riolo and B. Worzel, Eds., Boston, Springer, Boston, MA, 2005.

[9] E. Galvan and P. Mooney, “Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges,” *IEEE Transactions on Artificial Intelligence,* vol. 2, no. 6, pp. 476-493, December 2021.

[10] E. Real, C. Liang, D. R. So and Q. V. Le, “AutoML-Zero: Evolving Machine Learning Algorithms From Scratch,” 2020. [Online]. Available: https://arxiv.org/abs/2003.03384. [Accessed 9 July 2024].

[11] W. Tenachi, R. Ibata and F. I. Diakogiannis, “Deep symbolic regression for physics guided by units constraints: toward the automated discovery of physical laws,” *The Astrophysical Journal,* vol. 959, no. 2, p. 99, December 2023.

[12] Genetic And Evolutionary Computation, “Human-Competitive Awards 2004 – Present,” Genetic And Evolutionary Computation, [Online]. Available: https://www.human-competitive.org/. [Accessed 9 7 2024].

[13] J. R. Koza, “Human-competitive results produced by genetic programming,” *Genetic Programming and Evolvable Machines,* vol. 11, no. 3, pp. 251-284, 1 9 2010.

[14] T. Z. Caitlin Sadowski, Ed., “Removing Software Development Waste to Improve Productivity,” in *Rethinking Productivity in Software Engineering*, 1st ed., Apress Berkeley, CA, 2019, pp. 221-240.

[15] K. M. Adams, Introduction to Non-functional Requirements, 1 ed., Springer Cham, 2015, pp. 45-71.

[16] C. Ghezzi, M. Jazayeri and D. Mandrioli, Fundamentals of software engineering, 2 ed., Prentice-Hall, Inc., 1991.

[17] L. Abraham, T. Lestang, D. Wilby, F. Anderson and S. Lo, “Aim for understandability if you want to write good research software,” Software Sustainability Institute, 4 July 2022. [Online]. Available:

www.software.ac.uk/blog/aim-understandability-if-you-want-write-good-research-software. [Accessed 1 July 2024].

[18] S. Smith, M. Srinivasan and S. Shankar, "Debunking the myth that upfront requirements are infeasible for scientific computing software," in *Proceedings of the 14th International Workshop on Software Engineering for Science*, Montreal, Quebec, Canada, 2019.

[19] W. S. Smith and L. Lai, "A new requirements template for scientific computing," *Proceedings of the First International Workshop on Situational Requirements Engineering Processes–Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP,* vol. 5, no. 107-121, pp. 107-121, 2005.

[20] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM,* pp. 1053-1058, 11 1972.

[21] A. Shvets, Dive into Design Patterns,, 2019.

[22] M. R.-M. Kelly Rivers, "Functions," [Online]. Available: https://www.cs.cmu.edu/~15110-s20/slides/week2-2-functions.pdf. [Accessed 7 July 2024].

[23] “MicroPython,” [Online]. Available: github.com/micropython/micropython. [Accessed 2024].

[24] “Download the latest version of Python,” [Online]. Available: www.python.org/downloads/.

[25] L. Prechelt, “An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program.,” 2000.

[26] D. L. Parnas and P. Clements, “A Rational Design Process: How and Why to Fake it,” *IEEE Transactions on Software Engineering,* vol. 12, pp. 251-257, 2 1986.

[27] D. L. Parnas, P. Clement and D. M. Weiss, “The modular structure of complex systems,” *International Conference on Software Engineering,* pp. 408-419, 1984.

[28] PyTorch, “Build the Neural Network,” PyTorch, 2024. [Online]. Available: pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html. [Accessed 01 July 2024].

[29] D. Poulopoulos, “3 Keras Design Patterns Every ML Engineer Should Know,” Mmedium, 9 7 2021. [Online]. Available: https://towardsdatascience.com/3-keras-design-patterns-every-ml-engineer-should-know-cae87618c7e3. [Accessed 10 7 2024].

[30] H. Jin, “Design an Easy-to-Use Deep Learning Framework,” Medium, 10 April 2024. [Online]. Available: https://towardsdatascience.com/design-an-easy-to-use-deep-learning-framework-52d7c37e415f. [Accessed 3 7 2024].

[31] F. Chollet, “Keras API design guidelines,” 30 5 2020. [Online]. Available: https://github.com/keras-team/governance/blob/24401c1addf521e522fd363f6eb40e7c4c4881d5/keras_api_design_guidelines.md. [Accessed 4 7 2024].

[32] PyTorch, “Reproducibility,” [Online]. Available: https://pytorch.org/docs/stable/notes/randomness.html. [Accessed 15 8 2024].

[33] T. Blickle and L. Thiele, “A Comparison of Selection Schemes Used in Evolutionary Algorithms,” *Evolutionary Computation,* vol. 4, no. 4, pp. 361-394, 1996.

[34] P. J. B. Hancock, “An Empirical Comparison of Selection Methods in Evolutionary Algorithms,” in *Evolutionary Computing, AISB Workshop*, 1994.

[35] A. Agapie and A. H. Wright, “Theoretical Analysis of Steady State Genetic Algorithms,” *Applications of Mathematics,* vol. 59, no. 5, pp. 509-525, 10 2014.

[36] J. R. Koza, “Genetic programming as a means for programming computers by natural selection,” *Statistics and Computing,* vol. 4, no. 2, pp. 379-423, 1 6 1994.

[37] L. J. E. J. David Schaffer, “On Crossover as an Evolutionarily Viable Strategy,” *Proceedings of the Fourth International Conference on Genetic Algorithms,* pp. 61-68, 1991.

[38] E. Cantú-Paz, “A Survey of Parallel Genetic Algorithms,” [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a13fcba4fab4713d5acb2d970eddc6b148d2596d. [Accessed 8 8 2024].

[39] J. G. Falcón-Cardona, R. H. Gómez, C. A. C. Coello and M. G. C. Tapia, “Survey, Parallel Multi-Objective Evolutionary Algorithms: A

Comprehensive," *Swarm and Evolutionary Computation,* vol. 67, 2021.

[40] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau and C. Gagné, "Using Multiple Processors," 24 Jul 2023. [Online]. Available: https://deap.readthedocs.io/en/master/tutorials/basic/part4.html. [Accessed 14 8 2024].

[41] J. Blank, "Parallelization," 2020. [Online]. Available: https://pymoo.org/problems/parallelization.html. [Accessed 14 8 2024].

[42] Scikit-learn Developers, "Parallelism, resource management, and configuration," [Online]. Available: https://scikit-learn.org/stable/computing/parallelism.html. [Accessed 14 8 2024].

[43] S. Kim and J. Kang, "Optional: Data Parallelism," [Online]. Available: https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html. [Accessed 14 8 2024].

[44] E. Real, C. Liang, D. R. So and Q. V. Le, “AutoML-Zero: evolving machine learning algorithms from scratch,” in *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[45] S. Kelly, D. S. Park, X. Song, M. McIntire, P. Nashikkar, R. Guha, W. Banzhaf, K. Deb, V. N. Boddeti, J. Tan and E. Real, “Discovering Adaptable Symbolic Algorithms from Scratch,” in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Detroit, USA, 2033.

[46] M. F. Brameier and W. Banzhaf, Linear Genetic Programming, 1 ed., Springer Publishing Company, Incorporated, 2010.

[47] B. Doerr, C. Gießen, C. Witt and J. Yang, “The (1+λ) Evolutionary Algorithm with Self-Adjusting Mutation Rate,” *Proceedings of the Genetic and Evolutionary Computation Conference,* vol. 81, pp. 1351-1358, 1 July 2017.