

# Pyrimidine: An algebra-inspired Programming framework for evolutionary algorithms

Congwei Song  <sup>1</sup>

<sup>1</sup> Beijing Institute of Mathematical Sciences and Applications, Beijing, China

DOI: [10.21105/joss.06575](https://doi.org/10.21105/joss.06575)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: Sébastien Boisgérault 

Reviewers:

- [@mmore500](#)
- [@sjvrijn](#)

Submitted: 11 December 2023

Published: 01 November 2025

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

## Summary

**Pyrimidine** stands as a versatile framework designed for genetic algorithms (GAs), offering exceptional extensibility for a wide array of evolutionary algorithms.

Leveraging the principles of object-oriented programming (OOP) and the meta-programming, we introduce a distinctive design paradigm coined as “algebra-inspired Programming”, signifying the fusion of algebraic methodologies with the software architecture.

## Statement of need

GAs ([Holland, 1975](#); [Katoch et al., 2021](#)) have found extensive application across various domains and have undergone modifications and integrations with new algorithms ([Alam et al., 2020](#); [Cheng & Alkalifah, 2023](#); [Katoch et al., 2021](#)). For details about the principles of GA, refer to the references ([Holland, 1975](#); [Simon, 2013](#)).

In a typical Python implementation, populations are defined as lists of individuals, with each individual represented by a chromosome composed of a list of genes. Creating an individual can be achieved utilizing either the standard library’s array or the widely-used third-party library **NumPy** ([Harris et al., 2020](#)). The evolutionary operators are defined on these structures.

A concise comparison between pyrimidine and other frameworks is provided in [Table 1](#).

**Table 1:** Comparison of the popular genetic algorithm frameworks.

Library	Design Style	Versatil- ity	Extensi- bility	Visualiza- tion
<b>DEAP</b>	OOP, Meta- programming, Algebra-insprited OOP, Functional, Meta- programming	Universal	Extensi- ble	export the data in DataFrame
	OOP, decoration pattern	Universal	Limited by its philoso- phy	export the data in the class LogBook
	based on DEAP	Symbolic Regres- sion	Extensi- ble	Easy to Implement

Library	Design Style	Versatility	Extensibility	Visualization
<a href="#">tpot /gama</a>	<a href="#">scikit-learn</a> Style	Hyperparameter Optimization	Limited	-
<a href="#">gplearn/pysr</a>	scikit-learn Style	Symbolic Regression	Limited	-
<a href="#">scikit-opt</a>	scikit-learn Style	Numerical Optimization	Unextensible	Encapsulated as a data frame
<a href="#">scikit-optimize</a>	scikit-learn Style	Numerical Optimization	Very Limited	provide some plotting function
<a href="#">NEAT</a>	OOP	Neuroevolution	Limited	use the visualization tools

Tpot/gama ([Gijsbers & Vanschoren, 2021](#); [Olson et al., 2016](#)), gplearn/pysr, and scikit-opt follow the scikit-learn style ([Buitinck et al., 2013](#)), providing fixed APIs with limited extensibility. They are merely serving their respective fields effectively (including NEAT ([McIntyre et al., 2019](#))).

DEAP ([Fortin et al., 2012](#)) is feature-rich and mature. However, it adopts a tedious meta-programming style and some parts of the code lack decoupling, limiting its extensibility. Gaft is highly object-oriented and scalable, but inactive now.

Pyrimidine fully utilizes the OOP and meta-programming capabilities of Python, making the design of the APIs and the extension of the program more natural. So far, we have implemented a variety of optimization algorithms by pyrimidine, including adaptive GA ([Hinterding et al., 1997](#)), quantum GA ([Supasil et al., 2021](#)), differential evolution ([Radtke et al., 2020](#)), evolutionary programming ([Fogel & Fogel, 1986](#)), particle swarm optimization ([Wang et al., 2018](#)), as well as some local search algorithms, such as simulated annealing ([Kirkpatrick et al., 1983](#)).

To meet diverse demands, it provides enough encoding schemes for solutions to optimization problems, including Boolean, integer, real number types and their hybrid forms.

## Algebra-inspired programming

The innovative approach is termed “algebra-inspired Programming”. It should not be confused with so-called algebraic programming ([Kapitonova & Letichevskii, 1993](#)), but it draws inspiration from its underlying principles.

The advantages of the model are summarized as follows:

1. The population system and genetic operations are treated as an algebraic system, and genetic algorithms are constructed by imitating algebraic operations.
2. It is highly extensible. For example it is easy to define multi-populations, even so-called hybrid-populations.
3. The code is more concise.

## Basic concepts

We introduce the concept of a **container**, simulating an **(algebraic) system** where specific operators are not yet defined.

A container  $s$  of type  $S$ , with elements of type  $A$ , is represented by the following expression:

$$s = \{a : A\} : S \quad \text{or} \quad s : S[A], \quad (1)$$

where the symbol  $\{\cdot\}$  signifies either a set, or a sequence to emphasize the order of the elements. The notation  $S[A]$  mimicks Python syntax, borrowed from the module `typing`.

Building upon the concept, we define a population in pyrimidine as a container of individuals. The introduction of multi-population further extends this notion, representing a container of populations, referred to as “the high-order container”. Pyrimidine distinguishes itself with its inherent ability to seamlessly implement multi-population GAs.

An individual is conceptualized as a container of chromosomes, without necessarily being an algebraic system. Similarly, a chromosome acts as a container of genes.

In a population system  $s$ , the formal representation of the crossover operation between two individuals is denoted as  $a \times_s b$ , that can be implemented as the command `s.cross(a, b)`. Although this system concept aligns with algebraic systems, the current version diverges from this notion, and the operators are directly defined as methods of the elements, such as `a.cross(b)`.

The lifting of a function/method  $f$  is a common approach to defining the function/method for the system:

$$f(s) := \{f(a)\},$$

unless explicitly redefined. For example, the mutation of a population typically involves the mutation of all individuals in it. Other types of lifting are allowed.

transition is the primary method in the iterative algorithms, denoted as a transform:

$$T(s) : S \rightarrow S.$$

## Metaclasses

A metaclass should be defined to simulate abstract algebraic systems, which are instantiated as a set containing several elements, as well as operators and functions on them. Currently, the metaclass `MetaContainer` is proposed to create container classes without defining operators explicitly.

## Mixin classes

Mixin classes specify the basic functionality of the algorithm.

The `FitnessMixin` class is dedicated to the iteration process focused on maximizing fitness, and its subclass `PopulationMixin` represents the collective form.

When designing a novel algorithm, significantly differing from the GA, it is advisable to start by inheriting from the mixin classes.

## Base Classes

There are three base classes in pyrimidine: `BaseChromosome`, `BaseIndividual`, `BasePopulation`, to create chromosomes, individuals and populations respectively.

Generally, the algorithm design starts as follows.

```

class UserIndividual(MonoIndividual):
    element_class = BinaryChromosome

    def _fitness(self):
        # Compute the fitness

class UserPopulation(StandardPopulation):
    element_class = UserIndividual
    default_size = 10

```

Here, MonoIndividual represents an individual with a single chromosome. UserIndividual (or UserPopulation) serves as a container of elements in type of BinaryChromosome (or UserIndividual). Instead of setting the fitness attribute, users are recommended to override the \_fitness method, where the concrete fitness computation is defined. Following is the equivalent expression, using the notion in [Equation 1](#):

```

UserIndividual = MonoIndividual[BinaryChromosome]
UserPopulation = StandardPopulation[UserIndividual] // 10

```

Algebraically, there is no difference between MonoIndividual and Chromosome. And the population also can be treated as a container of chromosomes as follows.

```

class UserChromosome(BaseChromosome):
    def _fitness(self):
        # Compute the fitness

UserPopulation = StandardPopulation[UserChromosome] // 10

```

## References

- Alam, T., Qamar, S., Dixit, A., & Benaida, M. (2020). Genetic algorithm: Reviews, implementations, and applications. *CompSciRN: Computer Principles (Topic)*. <https://doi.org/10.22541/au.159164762.28487263>
- Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., & Varoquaux, G. (2013). API design for machine learning software: Experiences from the scikit-learn project. *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 108–122. <https://doi.org/10.48550/arXiv.1309.0238>
- Cheng, S., & Alkhalifah, T. (2023). Robust data driven discovery of a seismic wave equation. *Geophysical Journal International*, 236(1), 537–546. <https://doi.org/10.1093/gji/ggad446>
- Fogel, L. J., & Fogel, D. B. (1986). *Artificial intelligence through evolutionary programming: Prediction and identification* [Final Report]. U.S. Army Research Institute. <https://doi.org/10.21236/ada171544>
- Fortin, F.-A., Rainville, F.-M. D., Gardner, M.-A., Parizeau, M., & Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13, 2171–2175. <https://github.com/DEAP/deap>
- Gijssbers, P., & Vanschoren, J. (2021). GAMA: A general automated machine learning assistant. In Y. Dong, G. Ifrim, D. Mladenić, C. Saunders, & S. Van Hoecke (Eds.), *Machine learning and knowledge discovery in databases. Applied data science and demo track* (pp. 560–564). Springer International Publishing. [https://doi.org/10.1007/978-3-030-67670-4\\_39](https://doi.org/10.1007/978-3-030-67670-4_39)
- Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., ... Oliphant,

- T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
- Hinterding, R., Michalewicz, Z., & Eiben, A. E. (1997). Adaptation in evolutionary computation: A survey. *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, 65–69. <https://doi.org/10.1109/ICEC.1997.592270>
- Holland, J. (1975). *Adaptation in natural and artificial systems*. The Univ. of Michigan. <https://doi.org/10.7551/mitpress/1090.001.0001>
- Kapitonova, Y. V., & Letichevskii, A. A. (1993). Algebraic programming: Methods and tools. *Cybern Syst Anal*, 29, 307–312. <https://doi.org/10.1007/BF01125535>
- Katoch, S., Chauhan, S. S., & Kumar, V. (2021). A review on genetic algorithm: Past, present, and future. *Multimed Tools Appl*, 80, 8091–8126. <https://doi.org/10.1007/s11042-020-10139-6>
- Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220, 671–680. <https://doi.org/10.1126/science.220.4598.671>
- McIntyre, A., Kallada, M., Miguel, C. G., & Silva, C. F. da. (2019). Neat-python. *CodeReclaimers/Neat-Python*. <https://github.com/CodeReclaimers/neat-python>
- Olson, R. S., Urbanowicz, R. J., Andrews, P. C., Lavender, N. A., Kidd, L. C., & Moore, J. H. (2016). Automating biomedical data science through tree-based pipeline optimization. *Journal of Machine Learning Research*, 123–137. [https://doi.org/10.1007/978-3-319-31204-0\\_9](https://doi.org/10.1007/978-3-319-31204-0_9)
- Radtke, J. J., Bertoldo, G., & Marchi, C. H. (2020). DEPP - differential evolution parallel program. *Journal of Open Source Software*, 5(47), 1701. <https://doi.org/10.21105/joss.01701>
- Simon, D. (2013). *Evolutionary optimization algorithms: Biologically inspired and population-based approaches to computer intelligence*. John Wiley & Sons. <https://api.semanticscholar.org/CorpusID:60429433>
- Supasil, J., Pathumsoot, P., & Suwanna, S. (2021). Simulation of implementable quantum-assisted genetic algorithm. *Journal of Physics: Conference Series*, 1719(1), 012102. <https://doi.org/10.1088/1742-6596/1719/1/012102>
- Wang, D., Tan, D., & Liu, L. (2018). Particle swarm optimization algorithm: An overview. *Soft Computing*, 22(2), 387–408. <https://doi.org/10.1007/s00500-016-2474-6>