

1 Pyrimidine: An algebra-inspired Programming

2 framework for evolutionary algorithms




3 Congwei Song  ^{1*}

4 ¹ Beijing Institute of Mathematical Sciences and Applications, Beijing, China * These authors

5 contributed equally.

DOI: [10.xxxxxx/draft](#)


Software

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

6 Pyrimidine: An algebra-inspired Programming framework for

7 evolutionary algorithms

8 Summary

Editor: 

Submitted: 11 December 2023

Published: unpublished

License

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

9 [Pyrimidine](#) stands as a versatile framework designed for GAs, offering exceptional extensibility

10 for a wide array of evolutionary algorithms, including particle swarm optimization and difference

11 evolution.

12 Leveraging the principles of object-oriented programming(OOP) and the meta-programming, we

13 introduce a distinctive design paradigm is coined as “algebra-inspired Programming” signifying

14 the fusion of algebraic methodologies with the software architecture.

15 Statement of need

16 As one of the earliest developed intelligent algorithms [holland, katoch], the genetic al-

17 gorithm(GA) has found extensive application across various domains and has undergone

18 modifications and integrations with new algorithms ([Alam et al., 2020](#); [Cheng & Alkhalifah,](#)

19 [2023](#); [Katoch et al., 2021](#)). The principles of GA will not be reviewed in this article. For

20 a detailed understanding, please refer to references ([Holland, 1975](#); [Simon, 2013](#)) and the

21 associated literatures.

22 In a typical Python implementation, populations are initially defined as lists of individuals,

23 with each individual represented by a chromosome composed of a list of genes. Creating

24 an individual can be achieved utilizing either the standard library’s array or the widely-used

25 third-party library [numpy](#). Following this, evolutionary operators are defined and applied to

26 these structures.

27 ?? provides a concise comparison between pyrimidine and several popular frameworks, such as [DEAP](#)

28 Comparison of the popular genetic algorithm frameworks.

Library	Design Style	Versatility	Extensibility	Visualization
pyrimidine	OOP, Meta-programming, Algebra-inspired	Universal	Extensible	export the data in DataFrame
DEAP	OOP, Functional, Meta-programming	Universal	Limited by its philosophy	export the data in the class LogBook

Library	Design Style	Versatility	Extensibility	Visualization
gaft	OOP, decoration partton	Universal	Extensible	Easy to Implement
geppy	based on DEAP	Symbolic Regression	Limited	-
tpot (Olson et al., 2016)/gama (Gijssbers & Vanschoren, 2021)	scikit-learn Style	Hyperparameter Optimization	Limited	None
gplearn/pysr	scikit-learn Style	Symbolic Regression	Limited	None
scikit-opt	scikit-learn Style	Numerical Optimization	Unextendible	Encapsulated as a data frame
scikit-optimize	scikit-learn Style	Numerical Optimization	Very Limited	provide some plotting function
NEAT (McIntyre et al., n.d.)	OOP	Neuroevolution	Limited	use the visualization tools

29 Tpot/gama, gplearn/pysr, and scikit-opt follow the scikit-learn style (Buitinck et al., 2013),
30 providing fixed APIs with limited extensibility. They are merely serving their respective fields
31 effectively (as well as NEAT).

32 DEAP is feature-rich and mature. However, it primarily adopts a tedious meta-programming
33 style. Some parts of the source code lack sufficient decoupling, limiting its extensibility. Gaft
34 is a highly object-oriented software with excellent scalability, but it is currently inactive.

35 Pyrimidine fully utilizes the OOP and meta-programming capabilities of Python, making the
36 design of the APIs and the extension of the program more natural. So far, We have implemented
37 a variety of intelligent algorithms by pyrimidine, including adaptive GA (Hinterding et
38 al., 1997), quantum GA (Supasil et al., 2021), differential evolution (Radtke et al., 2020),
39 evolutionary programming, particle swarm optimization (Wang et al., 2018), as well as some
40 local search algorithms, such as simulated annealing.

41 Algebra-inspired programming

42 The innovative approach is termed “algebra-inspired Programming.” It should not be confused
43 with so-called algebraic programming, but it draws inspiration from its underlying principles.

44 Basic concepts

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

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

$$s = \{a : A\} : S$$

48 or equivalently

$$s : S[A]$$

49 where the symbol $\{\cdot\}$ signifies either a set, or a sequence to emphasize the order of the
50 elements, and the notation $S[\cdot]$ is borrowed from the module **typing** (*Typing — Support for
51 Type Hints, 2023*))

Building upon the foundational 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, often referred to as “the high-order container”. Pyrimidine distinguishes itself with its inherent ability to seamlessly implement multi-population GAs. Populations in a multi-population behave analogously to individuals in a population. Notably, it allows to define containers in higher order, such as a container of multi-populations, potentially intertwined with conventional populations.

While an individual can be conceptualized as a container of chromosomes, it will not necessarily be considered a system. Similarly, a chromosome might be viewed as a container of genes (implemented by the arrays in practice).

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 (*Algebraic Structure*, 2023), the current version of our framework 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, but there are cases where it may be defined as the mutation of a randomly selected individual. Another type of lifting is that the fitness of a population is determined as the maximum of the fitness values among the individuals in the population.

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

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

The iterative algorithms can be represented as $T^n(s)$.

Metaclasses

The metaclass `System` is defined to simulate abstract algebraic systems, which are instantiated as a set containing a set of elements, as well as operators and functions on them.

`Container` is the super-metaclass of `System` for creating containers.

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 and redefining the transition method, though it is not mandatory.

Base Classes

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

For convenience, pyrimidine provides some commonly used subclasses, where the genetic operations are implemented such as `cross` and `mutate`. Especially, pyrimidine offers `BinaryChromosome` for the binary encoding as used in the classical GA.

92 Generally, the algorithm design starts as follows, where `MonoIndividual`, a subclass of
 93 `BaseIndividual`, just enforces that the individuals can only have one chromosome.

```
class UserIndividual(MonoIndividual):
    element_class = BinaryChromosome
    # default_size = 1

    def _fitness(self):
        # Compute the fitness

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

94 In the codes, `UserIndividual` (resp. `UserPopulation`) is a container of elements in type of
 95 `BinaryChromosome` (resp. `UserIndividual`). Following is the equivalent expression(see ??):

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

96 Algebraically, there is no discrepancy between `MonoIndividual` and a single `Chromosome`. And
 97 the population also can be treated as a container of chromosomes. See the following codes.

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

UserPopulation = StandardPopulation[UserChromosome]
```

98 An example to begin

99 In this section, we demonstrate the basic usage of pyrimidine with the classic 0-1 knapsack
 100 problem, whose solution can be naturally encoded in binary format without the need for
 101 additional decoding:

$$\max \sum_i c_i x_i \quad \sum_i w_i x_i \leq W, \quad x_i = 0, 1, i = 1, \dots, n$$

```
from pyrimidine import BinaryChromosome, MonoIndividual, StandardPopulation
from pyrimidine.benchmarks.optimization import Knapsack
```

```
n = 50
_evaluate = Knapsack.random(n) # the objective function
```

```
class UserIndividual(MonoIndividual):
    element_class = BinaryChromosome // n
    def _fitness(self):
        return _evaluate(self[0])
```

```
"""
```

```
equivalent to:
```

```
UserIndividual = MonoIndividual[BinaryChromosome // n].set_fitness(lambda o: _evaluate(o))
"""
```

```
UserPopulation = StandardPopulation[UserIndividual] // 20
```

102 Using chromosome as the population's elements, we arrange all the components in a single
 103 line:

```
UserPopulation = StandardPopulation[BinaryChromosome // n].set_fitness(_evaluate)
```

104 Then we execute the evolutionary program as follows.

```
pop = UserPopulation.random()  
pop.evolve(n_iter=100)
```

105 Finally, the optimal individual can be obtained with `pop.best_individual`, or `pop.solution`
106 to decode the individual to the solution of the problem.

107 Visualization

108 Instead of implementing visualization methods, pyrimidine yields a `pandas.DataFrame` object
109 that encapsulates statistical results for each generation by setting `history=True` in `evolve`
110 method. Users can harness this object to plot the performance curves. Generally, users are
111 required to furnish a “statistic dictionary” whose keys are the names of the statistics, and values
112 are functions mapping the population to numerical values, or strings presenting pre-defined
113 methods or attributes of the population.

```
# statistic dictionary, computing the mean, the maximum and the standard deviation of the  
stat = {'Mean Fitness': 'mean_fitness',  
        'Best Fitness': 'max_fitness',  
        'Standard Deviation of Fitnesses': lambda pop: np.std(pop.get_all_fitness())  
}
```

```
# obtain the statistical results through the evolution.  
data = pop.evolve(stat=stat, n_iter=100, history=True)
```

```
import matplotlib.pyplot as plt  
fig = plt.figure()  
ax = fig.add_subplot(111)  
ax2 = ax.twinx()  
data[['Mean Fitness', 'Best Fitness']].plot(ax=ax)  
ax.legend(loc='upper left')  
data['Standard Deviation of Fitnesses'].plot(ax=ax2, style='y-.')  
ax2.legend(loc='lower right')  
ax.set_xlabel('Generations')  
ax.set_ylabel('Fitness')  
plt.show()
```

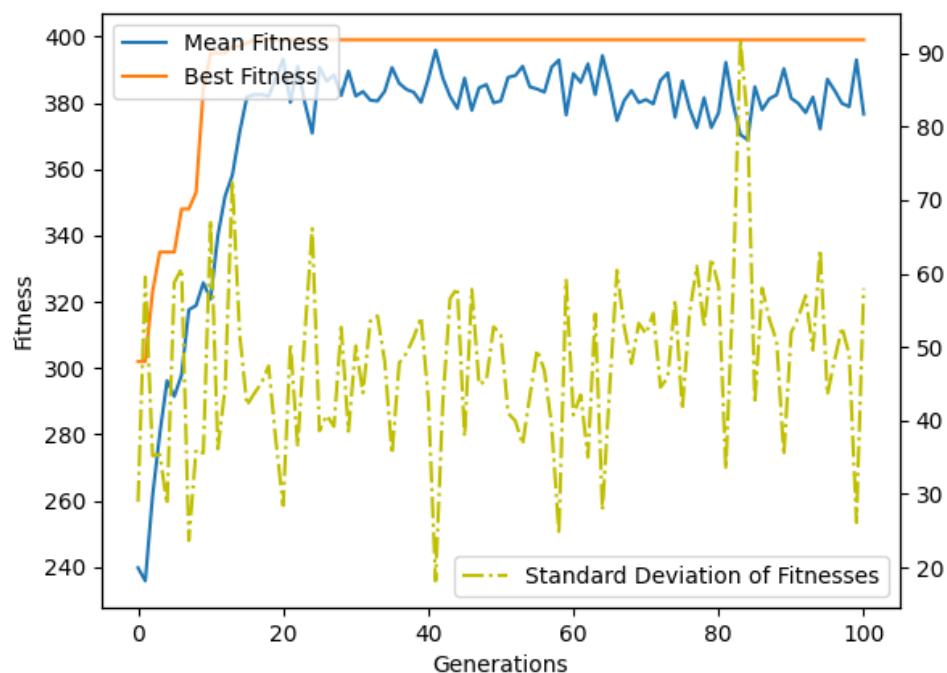


Figure 1: The fitness evolution curve of the population. The library does not provide specific plotting commands. Instead, users can directly utilize the plotting commands from the pandas library.

Conclusion

I have conducted extensive experiments and improvements, showcasing that pyrimidine is a versatile framework suitable for implementing various evolution algorithms. Its design offers strong extensibility, allowing the implementation of any iterative algorithm, such as simulated annealing or particle swarm optimization. For users developing new algorithms, pyrimidine is a promising choice.

We have not implemented parallel computation yet which is important for intelligent algorithms, but we have set up an interface that can be utilized at any time. The full realization of algebraic programming concepts is still in progress. The functionality of symbolic regression has not been realized yet, but we are considering reusing what DEAP provides rather than reinventing the wheel. Certainly, there is ample room for further improvement.

The source code has been uploaded to [GitHub](#), along with numerous examples.

References

- Alam, T., Qamar, S., Dixit, A., & Benaïda, M. (2020). Genetic algorithm: Reviews, implementations, and applications. *CompSciRN: Computer Principles (Topic)*. <https://api.semanticscholar.org/CorpusID:219914744>
- Algebraic structure*. (2023). https://en.wikipedia.org/wiki/Algebraic_structure
- 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.
- 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>
- Fortin, F.-A., Rainville, F.-M. D., Gardner, M.-A., Parizeau, M., & Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. In *Journal of Machine Learning Research* (Vol. 13, pp. 2171–2175).
- Gijsbers, P., & Vanschoren, J. (2021). GAMA: A general automated machine learning assistant. In Y. Dong, G. Ifrim, D. Mladenović, 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.
- 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.
- Katoch, S., Chauhan, S. S., & Kumar, V. (2021). A review on genetic algorithm: Past, present, and future. In *Multimed Tools Appl* (Vol. 80, pp. 8091–8126). <https://doi.org/10.1007/s11042-020-10139-6>
- McIntyre, A., Kallada, M., Miguel, C. G., Feher de Silva, C., & Netto, M. L. (n.d.). *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. In *Journal of Machine Learning Research* (pp. 123–137). 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.
- 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>
- Typing — support for type hints*. (2023). Python Software Foundation. <https://docs.python.org/3.11/library/typing.html?highlight=typing#module-typing>
- 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>