




pycellga: A Python package for improved cellular genetic algorithms

Sevgi Akten Karakaya ¹ and Mehmet Hakan Satman ²

¹ Department of Informatics, Istanbul University, Istanbul, Turkey ² Department of Econometrics, Istanbul University, Istanbul, Turkey

DOI: [10.21105/joss.07322](https://doi.org/10.21105/joss.07322)

Software

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

Editor: [Josh Borrow](#)  

Reviewers:

- [@jmejia8](#)
- [@jbussemaker](#)

Submitted: 14 August 2024

Published: 03 January 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

pycellga is a Python package that implements cellular genetic algorithms (CGAs) for optimizing complex problems. CGAs combine the principles of cellular automata and traditional genetic algorithms, utilizing a spatially structured population organized in a grid-like topology. This structure allows each individual to interact only with its neighboring individuals, promoting diversity and maintaining a balance between exploration and exploitation during the optimization process.

While CGAs themselves are not a novel contribution of this work, pycellga significantly enhances their applicability by integrating advanced features and providing unparalleled versatility. The package supports binary, real-valued, and permutation-based optimization problems, making it adaptable to a wide variety of problem domains. Its use of machine-coded operators for real-valued optimization, adhering to IEEE 754 floating-point arithmetic standards, ensures high precision and computational efficiency. Moreover, pycellga is designed to be extensible, enabling users to easily customize selection, crossover, and mutation operators to suit specific problem requirements.

The package is designed to be user-friendly, with a straightforward installation process and comprehensive documentation. Researchers and practitioners in fields such as operations research, artificial intelligence, and machine learning can leverage pycellga to tackle complex optimization challenges effectively. By integrating the principles of cellular automata with genetic algorithms, pycellga represents a significant advancement in the field of evolutionary computation, offering increased flexibility and adaptability compared to traditional methods.

Additionally, pycellga includes machine-coded operators with byte implementations, developed by Satman (2013). It features Alpha-male CGA, Machine-Coded Compact CGA, and Improved CGA with Machine-Coded Operators for real-valued optimization problems (Karakaya & Satman, 2024).

Introduction

Optimization problems are a fundamental aspect of various scientific and engineering fields, involving the search for the best solution among a large set of possible options. Genetic algorithms (GAs) have been widely used to address these problems due to their robustness and adaptability. Inspired by the process of natural selection, GAs operate on a population of potential solutions, applying operators such as selection, crossover, and mutation to evolve the population toward better solutions over successive generations (Goldberg, 1989; Holland, 1975).

Despite their effectiveness, traditional GAs face challenges, particularly in maintaining diversity within the population and avoiding premature convergence to suboptimal solutions (Goldberg

& Deb, 1991). To mitigate these issues, researchers have developed CGAs, which introduce a spatial structure to the population (Manderick & Spiessens, 1991; Whitley, 1993). In a CGA, individuals are placed on a grid, and interactions are restricted to neighboring individuals. This localized interaction promotes diversity and enables a more thorough exploration of the solution space.

`pycellga` is a Python package designed to efficiently implement CGAs. By integrating the principles of cellular automata with genetic algorithms, `pycellga` offers a robust framework for tackling complex optimization problems. The `pycellga` package is designed to handle a wide range of optimization problems, including binary, real-valued, and permutation-based challenges, making it a versatile tool for diverse applications in evolutionary computation. The package includes several built-in functions for initialization, selection, crossover, mutation, and evaluation, as well as customization options to cater to different needs. This flexibility allows researchers and practitioners to apply CGAs to a wide range of problems with ease (Karakaya & Satman, 2024).

By providing a comprehensive toolkit for CGAs, `pycellga` aims to advance the field of evolutionary computation and equip researchers with the tools needed to solve increasingly complex optimization problems effectively. The integration of cellular automata with genetic algorithms in `pycellga` represents a significant advancement, offering greater flexibility and adaptability compared to traditional methods (Eiben & Smith, 2015; Karakaya & Satman, 2024; Michalewicz, 1996).

The `pycellga` package includes machine-coded operators with byte-level implementations, developed by Satman (2013). In the context of genetic algorithms, “machine-coded” refers to a specialized encoding technique optimized for real-parameter optimization. This approach differs from standard coding practices by emphasizing efficient data processing through byte-level manipulation. Originally introduced by Satman (2013), this technique is particularly advantageous for real-valued optimization tasks, as it allows direct manipulation of byte-representations to enhance computational performance. Encoding and decoding of numerical values conform to the IEEE 754 standard for floating-point arithmetic, further improving precision and effectiveness in optimizing continuous functions. By using machine-coded operators, `pycellga` leverages this efficiency to handle complex optimization challenges more effectively.

In addition, the `pycellga` package features Alpha-male CGA, developed based on insights from Satman & Akadal (2019); Machine-Coded Compact CGA, inspired by Satman & Akadal (2020); and Improved CGA with Machine-Coded Operators (Karakaya & Satman, 2024). The improved cellular genetic algorithm utilizes machine-coded operators specifically tailored for real-valued optimization problems. This method is particularly distinctive for its use of byte-based operators, which are designed to process numerical data efficiently in terms of memory usage.

State of the field

There are several existing software packages that implement genetic algorithms, such as DEAP and PyGAD. These libraries provide a wide range of tools for evolutionary computation and are highly flexible for various optimization tasks. However, most of these packages do not specifically focus on the integration of cellular automata with genetic algorithms, except for JCell, which is a Java implementation of CGAs (Alba & Dorronsoro, 2008).

The lack of CGA variants in widely used Python libraries may be attributed to the complexity and niche appeal of cellular genetic algorithms. CGAs, by design, require a spatially structured population and localized interactions, which add an extra layer of complexity compared to traditional GAs. Additionally, while traditional GAs are well-documented and widely adopted, CGAs have been primarily explored in academic research, with fewer applications in

mainstream problem-solving scenarios. This may have led to limited adoption in general-purpose optimization libraries.

`pycellga` addresses this gap by offering a specialized toolkit for CGAs, leveraging the strengths of both cellular automata and genetic algorithms. By incorporating features such as Alpha-male CGA, Machine-Coded Compact CGA, and Improved CGA with Machine-Coded Operators, `pycellga` provides unparalleled support for tackling complex optimization problems. Its use of machine-coded operators, adhering to IEEE 754 floating-point arithmetic standards, ensures high precision and computational efficiency, making it a significant advancement in the field.

The introduction of `pycellga` represents a deliberate effort to bring CGA variants into broader usage, making these powerful algorithms more accessible to researchers and practitioners. By providing a Python-based implementation, `pycellga` bridges the gap between theoretical advancements in CGA research and their practical applications, thus addressing the limitations in existing software ecosystems.

Statement of need

The need for advanced optimization frameworks like `pycellga` arises from the growing complexity of real-world problems, which often involve high-dimensional search spaces, mixed-variable types, and intricate constraints. Traditional GAs, while highly versatile and widely adopted, face several limitations in addressing these challenges. These include a tendency toward premature convergence, difficulty in maintaining population diversity, and an inability to balance global exploration with local exploitation effectively (Goldberg, 1989; Holland, 1975).

CGAs offer a compelling solution by introducing a spatially structured population. In CGAs, individuals are arranged in a grid-like topology and interact only with their neighbors. This localized interaction not only enhances population diversity but also mitigates the risk of premature convergence, a common issue in traditional GAs (Manderick & Spiessens, 1991; Whitley, 1993). By promoting a gradual spread of genetic material through localized selection and crossover, CGAs achieve a more thorough exploration of the solution space, particularly in high-dimensional and multi-modal optimization problems (Alba & Dorronsoro, 2008). These attributes make CGAs especially effective for tackling complex challenges in scheduling, resource allocation, and multi-objective optimization (Carlos A. Coello Coello & Veldhuizen, 2007).

Despite the clear advantages of CGAs, their implementation in existing tools remains limited. Frameworks such as `JCell`, while functional, lack the flexibility, extensibility, and user-friendly features offered by modern programming environments like Python (Alba & Dorronsoro, 2008). To address this gap, `pycellga` provides a Python-based framework that combines the strengths of CGAs with the convenience and power of Python's ecosystem. The package includes efficient byte-level implementations of machine-coded operators, as introduced by Satman (2013), which significantly enhance performance for real-valued optimization problems. Additionally, it incorporates advanced CGA variants, such as Alpha-male CGA (Satman & Akadal, 2019), Machine-Coded Compact CGA (Satman & Akadal, 2020), and an Improved CGA with Machine-Coded Operators (Karakaya & Satman, 2024).

The unique contributions of `pycellga` extend beyond its robust implementation of CGAs. By supporting binary, real-valued, and permutation-based optimization problems, the package offers unparalleled versatility. Its use of machine-coded operators for real-valued optimization, adhering to IEEE 754 floating-point arithmetic standards, ensures high precision and computational efficiency. Moreover, `pycellga` is designed to be extensible, enabling users to easily customize selection, crossover, and mutation operators to suit specific problem domains.

In a field where traditional GAs have seen extensive research to address their inherent limitations (Eiben & Smith, 2015), CGAs provide a novel and effective approach to further these advancements. `pycellga` stands out as a modern, accessible, and feature-rich toolkit that

equips researchers and practitioners to tackle increasingly complex optimization challenges with confidence.

Installation and basic usage

`pycellga` can be downloaded and installed by using the following command:

```
pip install pycellga
```

Usage Examples

In this section, we'll explain the CGA method in the optimizer and provide an example of how to use it. The package includes various ready-to-use crossover and mutation operators, along with real-valued, binary, and permutation functions that you can run directly. Examples for other methods are available in the `example` folder, while an example for the CGA is provided below.

CGA

CGA is a type of genetic algorithm where the population is structured as a grid (or other topologies), and each individual interacts only with its neighbors. This structure helps maintain diversity in the population and can prevent premature convergence. To specialize the CGA for real-valued optimization problems, ICGA (Improved CGA) with machine-coded representation can be used, applying byte operators. The encoding and decoding of numbers follow the IEEE 754 standard for floating-point arithmetic, yielding better results for continuous functions.

Example Problem

Suppose we have a problem that we want to minimize using the CGA. The problem is defined as a simple sum of squares function, where the goal is to find a chromosome (vector) that minimizes the function.

The sum of squares function computes the sum of the squares of each element in the chromosome. This function reaches its global minimum when all elements of the chromosome are equal to 0. The corresponding function value at this point is 0.

ExampleProblem Class

Here's how we can define this problem in Python using the `ExampleProblem` class:

```
from mpmath import power as pw
from typing import List

from pycellga.optimizer import cga
from pycellga.recombination.byte_one_point_crossover import ByteOnePointCrossover
from pycellga.mutation.byte_mutation_random import ByteMutationRandom
from pycellga.selection.tournament_selection import TournamentSelection
from pycellga.problems.abstract_problem import AbstractProblem
from pycellga.common import GeneType

class ExampleProblem(AbstractProblem):

    def __init__(self, n_var):

        super().__init__(
            gen_type=GeneType.REAL,
```

```
n_var=n_var,  
xl=-100,  
xu=100  
)  
  
def f(self, x: List[float]) -> float:  
    return round(sum(pow(xi, 2) for xi in x),3)
```

Usage:

```
result = cga(  
    n_cols=5,  
    n_rows=5,  
    n_gen=100,  
    ch_size=5,  
    p_crossover=0.9,  
    p_mutation=0.2,  
    problem=ExampleProblem(n_var=5),  
    selection=TournamentSelection,  
    recombination=ByteOnePointCrossover,  
    mutation=ByteMutationRandom,  
    seed_par=100  
)  
  
# Print the results  
print("Best solution chromosome:", result.chromosome)  
print("Best fitness value:", result.fitness_value)
```

```
# Expected Output:  
# Best solution chromosome: [0.0, 0.0, 0.0, 0.0, 0.0]  
# Best fitness value: 0.0
```

References

- Alba, E., & Dorronsoro, B. (2008). *Cellular genetic algorithms*. Springer. https://doi.org/10.1007/978-0-387-77610-1_1
- Carlos A. Coello Coello, Gary B. Lamont, & Veldhuizen, D. A. V. (2007). *Evolutionary algorithms for solving multi-objective problems* (2nd ed.). Springer. <https://doi.org/10.1007/978-0-387-36797-2>
- Eiben, A. E., & Smith, J. E. (2015). *Introduction to evolutionary computing*. Springer. <https://doi.org/10.1007/978-3-662-44874-8>
- Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley. <https://doi.org/10.5860/choice.27-0936>
- Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms, 1*, 69–93. <https://doi.org/10.1016/b978-0-08-050684-5.50008-2>
- Holland, J. H. (1975). *Adaptation in natural and artificial systems*. MIT Press. ISBN: 9780262581110
- Karakaya, S. A., & Satman, M. H. (2024). An improved cellular genetic algorithm with machine-coded operators for real-valued optimisation problems. *Journal of Engineering Research and Applied Science, 13*(1), 2500–2514.
- Manderick, B., & Spiessens, P. (1991). The genetic algorithm and the structure of the fitness

- landscape. In R. K. Belew & L. B. Booker (Eds.), *Proceedings of the 4th international conference on genetic algorithms (ICGA)* (pp. 143–150). Morgan Kaufmann.
- Michalewicz, Z. (1996). *Genetic algorithms + data structures = evolution programs* (3rd ed.). Springer. <https://doi.org/10.1007/978-3-662-03315-9>
- Satman, M. H. (2013). Machine coded genetic algorithms for real parameter optimization problems. *Gazi University Journal of Science*, 26(1), 85–95.
- Satman, M. H., & Akadal, E. (2019). Performance comparison of the specialized alpha male genetic algorithm with some evolutionary algorithms. *Trakya University Journal of Social Science*, 21(1), 55–82. <https://doi.org/10.26468/trakyasobed.452095>
- Satman, M. H., & Akadal, E. (2020). Machine coded compact genetic algorithms for real parameter optimization problems. *Alphanumeric Journal*, 8(1), 43–58. <https://doi.org/10.17093/alphanumeric.576919>
- Whitley, D. (1993). Cellular genetic algorithms. In S. Forrest (Ed.), *Proceedings of the 5th international conference on genetic algorithms (ICGA)* (p. 658). Morgan Kaufmann.