Journal of open research software

StarPSO: A Unified Framework for Particle Swarm Optimization Across Multiple Problem Types

SOFTWARE METAPAPERS

MICHAIL D. VRETTAS
STEFANO SILVESTRI
*Author affiliations can be found in the back matter of this article

ubiquity press

ABSTRACT

StarPSO is an open source, object-oriented, Python library designed to implement various particle swarm optimization (PSO) algorithms, including: (i) StandardPSO, (ii) BinaryPSO, (iii) IntegerPSO, (iv) QuantumPSO, (v) CategoricalPSO, (vi) BareBonesPSO, and (vii) JackOfAllTradesPSO (for mixed variable type problems). This library addresses the challenge of optimizing diverse problem types through a unified framework. Implemented with performance optimizations using NumPy, Numba, and Joblib, it achieves efficient computation while preserving clean, well-documented, and maintainable code. Provided in a public GitHub repository, StarPSO encourages reuse and collaboration, allowing researchers and practitioners to easily integrate advanced optimization techniques into their own projects and benefit a wide range of applications across different domains.

CORRESPONDING AUTHOR:
Michail D. Vrettas
Researcher, Institute for High Performance Computing and Networking of National Research Council (ICAR-CNR), Via Pietro Castellino 111, 80131, Naples, Italy
michail.vrettas@gmail.com

KEYWORDS:
Swarm Intelligence; Evolutionary Computation; Particle Swarm Optimization; Multi-objective Problems; Constraint Optimization; Multimodal Problems; Python

TO CITE THIS ARTICLE:
Vrettas MD, Silvestri S 2026 StarPSO: A Unified Framework for Particle Swarm Optimization Across Multiple Problem Types. *Journal of Open Research Software*, 14: 38. DOI: https://doi.org/10.5334/jors.691

(1) OVERVIEW

INTRODUCTION

In 1995, James Kennedy and Russel Eberhart, inspired by simulations of simplified social models, introduced an optimization algorithm for nonlinear continuous functions that they dubbed "particle swarm optimization" (PSO) [1]. Drawing parallels to concepts from artificial life (e.g., bird flocking, fish schooling) and evolutionary computation (e.g., genetic algorithms (GAs) [2]), PSO has gained significant attention primarily due to its ease of understanding/explaining and straightforward coding implementation, which has led to the development of numerous versions of it [3].

PSO is a heuristic search technique characterized by its stochastic nature, which incrementally enhances a collection of candidate solutions, referred to as "swarm of particles," using a defined measure of fitness. The algorithm starts with an initial population of solutions, which are typically sampled at random, and seeks optimal solutions by iteratively adjusting the current positions of these particles. Unlike GAs where the solutions evolve, using a set of evolutionary operators, such as crossover and mutation, PSO solutions move through the search space by updating their positions in accordance with three distinct attractors: (1) the particle's current position, (2) the particle's personal historical best position, which serves as a cognitive convergence point, and (3) the best position identified by the entire swarm, functioning as a social convergence point.

The mechanics of the original PSO are remarkably simple and can be described by two straightforward equations. Equation (1) represents the core mechanism for updating the particle's velocity. It defines how i^{th} particle's future velocity $\mathbf{v}_i^{t+1}$ (at iteration $t+1$), is influenced by its current velocity $\mathbf{v}_i^t$, its own personal best known position $\mathbf{p}_i$, and the global best-known position $\mathbf{g}_{best}$, together with some random influences represented by two uniform random vectors $\mathbf{u}_1$ and $\mathbf{u}_2$. This dynamic allows particles to explore the search space while being attracted to both their own and the global best positions. Subsequently, Eq. (2) computes the new position $\mathbf{x}_i^{t+1}$ of the i^{th} particle by adding its updated velocity to its current position $\mathbf{x}_i^t$.

$$\mathbf{v}_i^{t+1} := w \cdot \mathbf{v}_i^t + \mathbf{u}_1 \circ (\mathbf{p}_i - \mathbf{x}_i^t) + \mathbf{u}_2 \circ (\mathbf{g}_{best} - \mathbf{x}_i^t), \quad (1)$$

$$\mathbf{x}_i^{t+1} := \mathbf{x}_i^t + \mathbf{v}_i^{t+1}, \quad (2)$$

where boldface letters $\mathbf{v}_i$, $\mathbf{p}_i$, $\mathbf{g}_{best}$, $\mathbf{x}_i \in \Re^d$ and $\mathbf{u}_1 \sim \mathcal{U}(0, c_1)^d$, $\mathbf{u}_2 \sim \mathcal{U}(0, c_2)^d$ represent d-dimensional vectors, while w, c_1, $c_2 \in \Re$, denote scalar values. Symbol "$\cdot$" represents normal multiplication, in contrast to "$\circ$" that represents an element-wise multiplication between two vectors of the same size.

Existing Python implementations of the basic/standard PSO can be found in some evolutionary algorithm toolboxes, such as DEAP [4], and other frameworks that offer single- and multi-objective optimization algorithms, such as pymoo [5]. Alternatively, a PSO-specific library that implements the classic PSO algorithm (global and local best) is provided by PySwarms [6].

StarPSO is an advanced research toolbox, specifically designed to implement a range of PSO algorithms. It addresses problems involving continuous, discrete, categorical, and mixed variable types within a single unified framework. This library effectively handles the complexities associated with optimizing diverse problem types by providing a cohesive platform. Furthermore, its clearly defined and well-structured object-oriented design facilitates the integration of new PSO algorithms, enhancing its adaptability and utility in research applications.

IMPLEMENTATION AND ARCHITECTURE

StarPSO is implemented in Python programming language which, according to the 2026 TIOBE (programming community) index [7] and the Stack Overflow Developer Survey [8], is the most dominant in several fields of computer science, such as artificial intelligence (AI) and machine learning (ML), due to its ease of learning, rapid development (fast prototyping), rich ecosystem, and cross-platform compatibility. The code is organized in four main packages:

1. **[engines]**: this package includes the computational models of all the available PSO implementations. Additional engines can be added in here, after inheriting from the GenericPSO class, which provides the basic functionality and acts as the required interface.
2. **[population]**: this package contains the main modules that are necessary to construct and represent the swarm of solutions to any optimization problem (i.e., the Particle and Swarm classes). Moreover, since the "JackOfAllTrades" algorithm requires specialized data structures to represent the mixed variable types, for this engine, the particle is modeled by the specific JatParticle class.
3. **[utils]**: includes two modules: (i) auxiliary and (ii) data_block. The former contains a list of supplementary data structures (e.g., BlockType), utility decorators (e.g., @time_it, @cost_function), along with a list of numba-jit optimized numerical functions that are used frequently by all the computational models (i.e., engines). While the later is used solely by the "JackOfAllTrades" version and contains the main class that encodes the data of a single optimization variable (i.e., a DataBlock), along with its core functionality that allows the update of

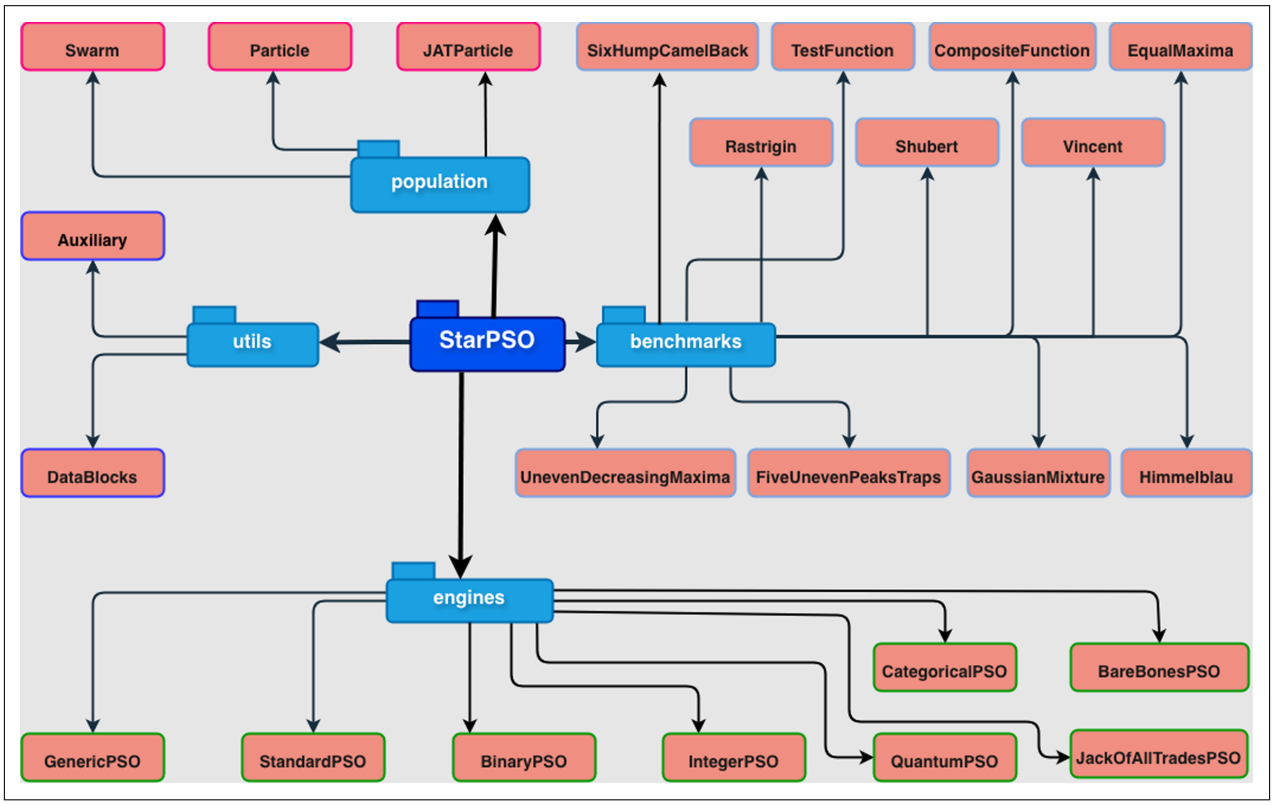

Figure 1 An overview of the StarPSO code structure. The four main packages are represented by the light blue tabbed ovals, while the modules inside each package are represented by light red ovals, with different edge color.

particle's position, according to its specific block type (i.e., float, int, binary, or categorical).

4. **[benchmarks]**: a particular strength of StarPSO is its built-in ability to be applied to multimodal problems (i.e., optimization problems where there exists more than one optimal value). This package contains a collection of 10 multimodal test functions [9], which vary in dimensionality from 1D (such as EqualMaxima, FiveUnevenPeakTrap) to 2D (e.g., Himmelblau, SixHumpCamelBack), and extends to dimensions greater than 2D (e.g., functions like Rastrigin and Shubert). More benchmark functions can be added by inheriting directly from the TestFunction class, which offers the essential functionality and serves as the necessary interface.

A graphical representation of the code structure is given in Figure 1.

SOFTWARE FUNCTIONALITIES

The current version of StarPSO implements a variety of PSO algorithms that can be used on problems with different data types. We have to note that since the original PSO algorithm has undergone various transformations through the years, there exists a plethora of different variations. Therefore, we do not claim that there is a right or wrong version, but rather that we provide a set of implementations that they can be used as is, or tailored to match specific requirements.

1. **Standard:** this version is the most common in practice, and it is used for continuous space domains. It is based on the work of [10] with the introduction of the *inertia weight* parameter in the original particle swarm optimizer [1].
2. **Binary:** operates on discrete binary variables following [11], where the velocities of the particles are functioning as probability thresholds.
3. **Integer:** provides a similar workflow as the Standard, but it rounds the positional variables, of the particles, to the nearest integer (dubbed IPSO in [12]). For positions exactly halfway between rounded decimal values, the method rounds to the nearest even integer value.
4. **Categorical:** addresses problems, discrete in nature, with variables that indicate categories without a specific order (i.e. nominal variables). It provides a simpler alternative to the ICPSO algorithm [12], which incorporates ideas from estimation of distribution algorithms (EDAs) in that particles' positions represent probability distributions rather than solution values.
5. **Quantum:** provides a version of the weighted QPSO (WQPSO) algorithm [13], which belongs to the family of *quantum-behaved* PSO algorithms. Quantum-behaved algorithms integrate principles of quantum mechanics with traditional PSO methods. These algorithms aim to improve the search for global optimal values in various optimization problems, with potentially faster convergence, compared to the Standard PSO.

6. **BareBones:** this algorithm is used for the optimization of continuous variables using the *bare-bones* PSO rules, where the velocity updates have been completely removed and the new position updates are sampled directly by a Gaussian distribution centered on the particle's personal best and the swarm/ global best [14]. This simplified variant of the original PSO preserves the cooperative search behavior of the swarm, while reducing algorithmic complexity and hyperparameters, often yielding competitive performance on continuous optimization tasks.
7. **JackOfAllTrades:** introduces a new PSO implementation able to apply the original equations (as in the Standard), but for mixed variable problems. This is achieved by considering the particle as a collection (or container) of data blocks, where each data block represents a single optimization variable and can belong to one of the following types: (i) float, (ii) integer, (iii) binary, or (iv) categorical. Subsequently, the algorithm follows the same process and when it comes to the step of updating the positions of the particles, each data block calls internally its own function of updating its positional variables. In other words, the data block structure not only holds the correct type of the positional variables, but also encapsulates the correct update functionality.

Furthermore, all the above algorithms provide support for the following options (for a summary see Table 1):

Adaptive: Contrary to GAs, which require a high number of parameters to be tuned for optimal performance [15] (e.g., choice of selection, crossover, mutation operators, probability values, number of elite population, etc.), PSO algorithms usually require only a handful of adjustable parameters, such as the inertia weight "w," along with the cognitive "c_1" and social "c_2" parameters (see Eq. 1). Even though there are only three parameters, their role in balancing the trade-off between exploration and exploitation in the algorithm is crucial, therefore they can have a significant impact in the final result. StarPSO offers the option to allow these parameters to adapt, during runtime, based on the convergence of the population to a single solution. Each of the different implementations has a measure of the spread of the solutions in the search space, and if the spread is small (indicating the convergence to a single solution), the algorithm adjusts slightly the social and cognitive coefficients to allow the particles to explore a bit further for a better solution, or vice versa.

Mode: Unlike the previous adaptive parameter that is either enabled or disabled, the "mode" parameter is always set to one of its three options: (i) global best (G_{best}), (ii) fully informed (FI), and (iii) multimodal. This parameter affects directly the $\mathbf{g}_{best}$ vector in the velocity equations (1), which deals with the social attractor of the particle. By default, its value is set to G_{best}, which simply seeks the particle, from the whole swarm, with the optimal fitness value. The fully informed option is based in [16] and sets as a global attractor the weighted average of the particles' best positions. Finally, the "multimodal" option is a new technique that enables the swarm to simultaneously concentrate on multiple optimal values instead of converging on just one solution. This is the preferred mode when we are dealing with multimodal problems, and we want to identify more than one optimal solutions, with a single run of the algorithm.

Parallel: Population based algorithms are inherently parallelizable. That means there is always a part of the algorithm that can be computed independently for each member of the population. As PSO algorithms fall in the same category, there are several parts of them, that can be executed in parallel. However, Python is known to have some limitations in multithreaded execution, mainly due to the global interpreter lock (GIL), which is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecode simultaneously. Therefore, StarPSO offers the parallel option in a multiprocessing framework (using JobLib [17]), by allowing the fitness function of the swarm to be evaluated in several central processing units (CPUs).

ALGORITHM	VARIABLE TYPE	OPTIONS		
		ADAPTIVE	MODE	PARALLEL
Standard	Continuous	✓	[G_{best}], FI, Multimodal	✓
Binary	Discrete	✓	[G_{best}], FI	✓
Integer	Discrete	✓	[G_{best}], FI	✓
Categorical	Categorical	✓	[G_{best}], FI	✓
Quantum	Continuous	–	[G_{best}], FI, Multimodal	✓
BareBones	Continuous	–	[G_{best}], FI, Multimodal	✓
JackOfAllTrades	Mixed	✓	[G_{best}], FI	✓

Table 1 A summary of the available options for the different StarPSO implementations. The default value for the "Mode" is marked with square brackets.

As with every iterative optimization algorithm that is required to have at least one termination condition, to avoid running perpetually; StarPSO provides four termination conditions, as seen in Table 2.

CONDITION	TYPE	DEFAULT VALUE	OPTIONAL
Maximum number of iterations	Integer	`1000`	No
Maximum fitness evaluations	Integer	`None`	Yes
Fitness (convergence) tolerance	Float	`None`	Yes
Found solution	Boolean	`False`	Yes

Table 2 Types and default values of the termination conditions. The optional numeric conditions default to None.

The first condition sets a maximum number of iterations for the algorithm to update its particles positions. This acts as an upper bound and ensures that if no other condition is set (or met with) the algorithm will terminate. In environments where there are limitations on the resources, StarPSO also allows the user (optionally) to set a maximum number of fitness function evaluations. This feature is also useful when one compares two algorithms and wants to make sure that both algorithms use the same computational resources. Another optional termination condition is setting a tolerance threshold value for the improvement of the fitness function. This is usually set to a very small value (e.g. $f_{tol} = 10^{-8}$) and terminates the algorithm if the gains in the fitness are small. Moreover, in various optimization problems, it is possible to verify that a solution satisfies some termination criteria. In these cases, StarPSO can make direct use of that condition to signal the algorithm for termination. This feature requires the user to explicitly return a Boolean variable (flag), along with the computed fitness values.

Finally, all algorithms come with default values that would satisfy many optimization problems and allow the methods to run even without passing a single parameter. Nevertheless, the user is encouraged to set new parameter values depending on the problem at hand.

ILLUSTRATIVE EXAMPLES

To demonstrate a step-by-step example of the StarPSO library, we use the Himmelblau function [18]. This is a two-dimensional multimodal function that tests the performance of optimization algorithms. Here, we present a modified (inverted) version of the function defined as follows [9]:

$$f(x, y) = 200 - (x^2 + y - 11)^2 - (x + y^2 - 7)^2, \qquad (3)$$

where $x, y \in \Re$. It has four global optima with two of them closer to each other, as can be seen in Figure 2.

As shown in Listing 2, the basic steps that are required are as follows:

1. By definition, the PSO algorithms require a measure of fitness. This is specific to the optimization problem that is being solved and identifies which solutions are better than others. As a rule of thumb, this function will have to follow the template as seen in Listing 1.

```python
@cost_function(minimize=False)
def objective_fun(x_in: NDArray,
                  **kwargs) -> tuple[float, bool]:
# ... CODE TO IMPLEMENT ...

# Compute the function value.
f_value =  ...

# Condition for termination.
solution_found = ...

return f_value, solution_found

```

Listing 1 Objective function template.

   Note that if the optimization problem refers to a *minimization* or a *maximization*, it has to be declared explicitly in the custom decorator `@cost_function`. Moreover, if a condition for termination is not possible to be verified, the user can simply omit the return of the second variable (`solution_found`).
2. The initialization of the swarm is usually done by sampling randomly within the limits of the search domain. Alternatively, StarPSO allows the user to initialize the population from a predefined swarm. This can be useful in cases where the optimization process has to stop (e.g., due to limited allocation of resources) and then re-start from the same point that it stopped.
3. Depending on the problem that one has to solve, the selection of an appropriate PSO implementation is crucial. For example, if one has to solve a problem in continuous domain, something that is addressed by the StandardPSO, selecting the BinaryPSO, or IntegerPSO could be the wrong choice.
4. Finally, once the initial swarm and the correct PSO have been created, the next step is to simply run the method. As mentioned earlier, the algorithms have default parameter values that would suffice for most cases, but using prior (problem-specific) knowledge to customize them is highly recommended. One thing to pay attention in multimodal cases is that the adaptation of the parameters should be disabled. The reason is that the current implementation uses the spread of the swarm as a measure of convergence. However, in problems with multiple optimal solutions, we want the swarm not to focus on a specific area, but rather explore all the domain to find all the possible solutions. This way, the spread of the whole swarm would work counter-intuitively.

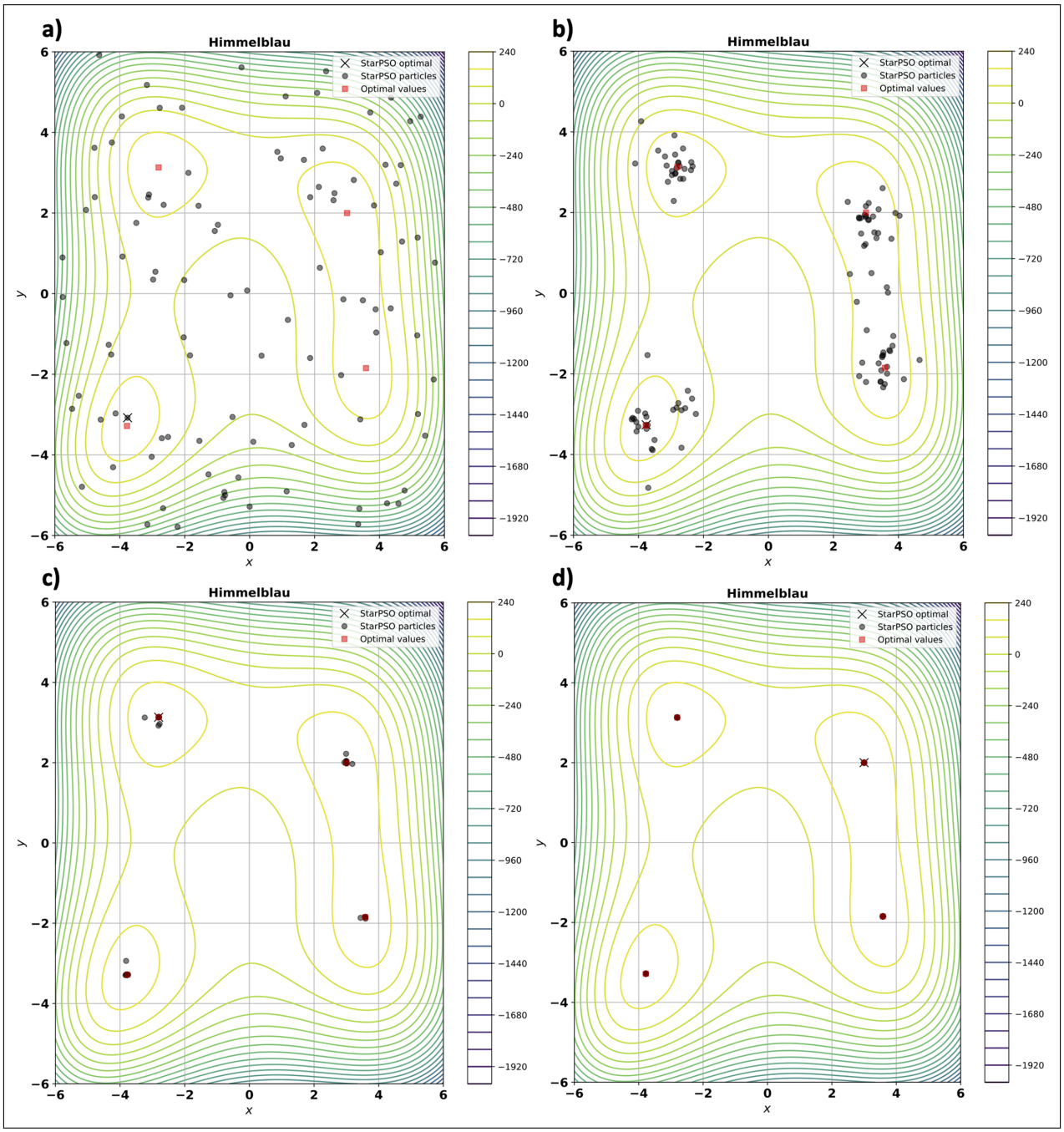

Figure 2 Contour plot of the Himmelblau 2D function at different iterations: **(a)** at initialization, **(b)** after 10 iterations, **(c)** after 50 iterations, and **(d)** at convergence after 200 iterations. All of the four modes have been successfully identified by the swarm particles. The symbol "×" marks the location of the particle with the optimal fitness value.

A comprehensive list of examples on how to use all the available implementations on various problems involving: (i) single/multiple objective(s), (ii) single/multiple mode(s), and (iii) using constraints (inequalities) can be found in the online documentation as well as the project's GitHub page.

```python
# Import numpy type alias.
from numpy.typing import NDArray

# Import StarPSO classes.
from star_pso.population.swarm import Swarm
from star_pso.population.particle import Particle
from star_pso.utils.auxiliary import cost_function
from star_pso.engines.standard_pso import StandardPSO

# Import a (test) benchmark function.
from star_pso.benchmarks.himmelblau import Himmelblau

# Create a test problem.
benchmark = Himmelblau(x_min=-6.0, x_max=6.0)

# STEP 1: Define the objective function.
@cost_function(minimize=False)
def objective_fun(x_in: NDArray, **kwargs) -> float:
    return benchmark.func(x_in).item()
# _end_def_

# STEP 2.1: Get an initial (random) sample.
x_sample = benchmark.sample_random_positions(n_pos=100)

# STEP 2.2: Initialize the swarm population.
swarm_t0 = Swarm([Particle(x) for x in x_sample])

# STEP 3: Create the StandardPSO object.
```

```
test_PSO = StandardPSO(
           initial_swarm=swarm_t0,
           obj_func=objective_fun,
           x_min=benchmark.x_min,
           x_max=benchmark.x_max)

# STEP 4: Run the optimization.
test_PSO.run(max_it=200,
             options={"w0": 0.70,
                      "c1": 1.50,
                      "c2": 1.50,
                      "mode": "multimodal"})

> Output:
> Final f_optimal = 200.000
> run: elapsed time = 0.835 seconds.
```

Listing 2 Simple multimodal example using the Himmelblau 2D function.

QUALITY CONTROL

Ensuring the quality of the code is of high priority for this project, as it directly influences the overall performance, maintainability, and user satisfaction of our software. We use an object-oriented programming paradigm to structure our code, which promotes clean design and modularity. To achieve high standards of code quality, we adhere to the following best practices:

- **Thoroughly documented code**: Comprehensive documentation serves as a vital resource for researchers, developers, and end-users, enabling them to grasp the code's purpose, underlying logic, and structural organization. Additionally, our API documentation is readily available online, offering real-time access to essential information that aids in effective utilization of our algorithm's features.
- **Consistent coding style**: Maintaining a uniform coding style across the project significantly enhances code readability. A consistent approach not only makes the code easier to read and navigate but also facilitates smoother collaboration among team members.
- **Minimal code complexity**: Our aim is to produce code that is straightforward and easy to comprehend. By minimizing complexity, we not only create less convoluted code but also enhance its efficiency. Simplified code can lead to faster execution, which ultimately improves the overall system performance.
- **Unit tests**: Implementing adequate unit tests is crucial for early identification of bugs within the development lifecycle. By catching issues early, we can significantly reduce the time associated with fixes later in the process. Unit tests provide an assurance that individual components of the software perform as intended, contributing to the reliability of the overall system.

Moreover, to ensure adherence to coding standards and best practices, we use Pylint [19], a popular static code analysis tool, which scored StarPSO with: 9.50/10. This high score reflects our commitment to maintaining high code quality, readability, and alignment with Python conventions.

To ensure that we consistently meet these high standards, we will conduct regular reviews and checks. These evaluations will help us maintain our coding practices and adapt to any changes or improvements within the programming landscape, ensuring that our project remains robust and capable of adapting to future requirements. Through a continuous commitment to quality control, we aim to deliver a product that meets the highest expectations of performance and reliability.

(2) AVAILABILITY

OPERATING SYSTEM

StarPSO package is platform-independent and it can be run on any operating system (e.g., GNU/Linux, Mac OSX, Windows) that supports Python (version 3.10 or greater). Current operating systems that have been tested on include: macOS Catalina 10.15.7. and macOS Ventura 13.7.8.

PROGRAMMING LANGUAGE

Python (version 3.10 or greater).

ADDITIONAL SYSTEM REQUIREMENTS

None

DEPENDENCIES

The current version has been developed and tested with the following packages.

- Required:
  - numpy: 2.1.3
  - numba: 0.61.0
  - joblib: 1.4.2
  - scipy: 1.15.3 (for benchmark functions)
- Optional (for running the example notebooks):
  - jupyter: 7.3.2
  - matplotlib: 3.10.0
  - sphinx_rtd_theme: (for building the doc files)

LIST OF CONTRIBUTORS

Michail D. Vrettas is the primary architect, developer and current maintainer of the StarPSO package. Stefano Silvestri advised and co-authored this article.

SOFTWARE LOCATION

Archive

Name: Zenodo

Persistent identifier: https://doi.org/10.5281/zenodo.18429187

Licence: GNU General Public License v3.0 or later
Publisher: Michail D. Vrettas
Version published: v.0.1.2
Date published: 30/01/2026

Code repository

Name: GitHub
Persistent identifier: https://github.com/vrettasm/PyStarPSO
Licence: GNU General Public License v3.0 or later
Date published: 30/01/2026

LANGUAGE

English

(3) REUSE POTENTIAL

PSO algorithms have been used in various scientific research and industrial fields, with applications in electrical and electronic engineering, automation control systems, communication theory, operations research, mechanical engineering, fuel and energy, medicine, chemistry, and biology. A comprehensive survey is given in [20], and references therein.

StarPSO comes with support for the most frequently used PSO algorithms; however, this collection of algorithms can be extended by providing new "engines" that inherit directly the *GenericPSO* class. Moreover, the swarm update rules can be modified directly using the two relevant methods: (1) update_velocities and (2) update_positions. These two methods are part of the general interface and each PSO has to implement them accordingly. The *GenericPSO* class, which serves as the base for all subsequent algorithms, implements the original formulation of Eq. (1). Algorithms that use different update rules—such as *QuantumPSO* and *BareBonesPSO*—must provide their own implementations, overriding the default. Information on how to extend the software is available both through the API docs and example notebooks.

The software is maintained on GitHub under an open-source license (GPL-3.0). Users can report bugs, request features, or seek support by opening an issue on the repository. Alternatively, users may contact the primary author directly via email for inquiries related to the software. This method is in line with the principles of open science, highlighting transparency, accessibility, collaboration, and reproducibility, which fosters its broad acceptance among researchers.

FUNDING STATEMENT

This work is supported by the European Commission, DIGITAL-2021-SKILLS-01-SPECIALISED programme, grant number 101083563, project "DS4Health – Digital Skills for Healthcare Transformation."

COMPETING INTERESTS

The authors have no competing interests to declare.

AUTHOR AFFILIATIONS

Michail D. Vrettas orcid.org/0000-0002-5456-3226
Researcher, Institute for High Performance Computing and Networking of National Research Council (ICAR-CNR), Via Pietro Castellino 111, 80131, Naples, Italy

Stefano Silvestri orcid.org/0000-0002-9890-8409
Researcher, Institute for High Performance Computing and Networking of National Research Council (ICAR-CNR), Via Pietro Castellino 111, 80131, Naples, Italy

REFERENCES

1. **Kennedy J, Eberhart R.** Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume 4. IEEE; 1995. pp. 1942–1948. DOI: https://doi.org/10.1109/ICNN.1995.488968
2. **Holland JH.** *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press; 1975.
3. **Poli R, Kennedy J, Blackwell T.** Particle swarm optimization: An overview. *Swarm Intelligence*. 2007;1: 33–57. DOI: https://doi.org/10.1007/s11721-007-0002-0
4. **Félix-Antoine F, De Rainville F-M, Gardner M-A, Parizeau M, Gagné C.** DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*. 2012;13:2171–2175.
5. **Blank J, Deb K.** pymoo: Multi-Objective Optimization in Python. *IEEE Access*. 2020;8:89497–89509. DOI: https://doi.org/10.1109/ACCESS.2020.2990567
6. **Miranda LJV.** PySwarms: a research toolkit for Particle Swarm Optimization in Python. *The Journal of Open Source Software*. 2018;3(21):433. DOI: https://doi.org/10.21105/joss.00433
7. **TIOBE (the software quality company).** *TIOBE Index*. March 2026. https://www.tiobe.com/tiobe-index/
8. **Stack Overflow.** *Stack Overflow Developer Survey*. 2025. https://survey.stackoverflow.co/2025/
9. **Li X, Engelbrecht A, Epitropakis MG.** Benchmark Functions for CEC'2013 Special Session and Competition on Niching Methods for Multimodal Function Optimization. Report, Evolutionary Computation and Machine Learning Group, RMIT University. 2013.
10. **Shi Y, Eberhart R.** A modified particle swarm optimizer. In *Proceedings of the IEEE World Congress on Computational Intelligence*, 1998. pp. 69–73. DOI: https://doi.org/10.1109/ICEC.1998.699146
11. **Kennedy J, Eberhart R.** A discrete binary version of the particle swarm algorithm. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 5 of *Computational Cybernetics and Simulation*. IEEE; 1997. pp. 4104–4108. DOI: https://doi.org/10.1109/ICSMC.1997.637339
12. **Strasser S, Goodman R, Sheppard J, Butcher S.** A new discrete particle swarm optimization algorithm. In

Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2016. Denver, Colorado, USA: Association for Computing Machinery; 2016. pp. 53–60. DOI: https://doi.org/10.1145/2908812.2908935

13. **Xi M, Sun J, Xu W.** An improved quantum-behaved particle swarm optimization algorithm with weighted mean best position. *Applied Mathematics and Computation*, 2008; 205(2):751–759. DOI: https://doi.org/10.1016/j.amc.2008.05.135
14. **Kennedy J.** Bare bones particle swarms. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium SIS'03 (Cat. No.03EX706)*. Indianapolis, IN, USA: IEEE; 2003. pp. 80–87. DOI: https://doi.org/10.1109/SIS.2003.1202251
15. **Vrettas MD, Silvestri S.** PyGenAlgo: A simple and powerful toolkit for genetic algorithms". *SoftwareX*, 2025;30:102127. ISSN 2352-7110. DOI: https://doi.org/10.1016/j.softx.2025.102127
16. **Mendes R, Kennedy J, Neves J.** The fully informed particle swarm: Simpler, maybe better. *IEEE Transactions on Evolutionary Computation*, 2004;8(3):204–210. DOI: https://doi.org/10.1109/TEVC.2004.826074
17. **Joblib contributors.** *Joblib: running Python functions as pipeline jobs*. 2025. https://github.com/joblib/joblib. Latest release: 1.5.3.
18. **Himmelblau DM.** *Applied Nonlinear Programming*. New York: McGraw-Hill, first edition; 1972. ISBN 978-0070289215.
19. **Pylint contributors.** *Pylint: The Python static code analysis tool*. 2025. https://github.com/pylint-dev/pylint. Stable release: 4.0.4.
20. **Zhang Y, Wang S, Ji G.** A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications. *Mathematical Problems in Engineering*. 2015. DOI: https://doi.org/10.1155/2015/931256

TO CITE THIS ARTICLE:
Vrettas MD, Silvestri S 2026 StarPSO: A Unified Framework for Particle Swarm Optimization Across Multiple Problem Types. *Journal of Open Research Software*, 14: 38. DOI: https://doi.org/10.5334/jors.691

Submitted: 02 February 2026 **Accepted:** 12 May 2026 **Published:** 22 May 2026