VAMOS: A Vectorized Architecture for Multiobjective Optimization Studies
A High-Performance Python Framework with Adaptive and Auto-Configurable Components

First Author[a,*], Second Author[a]

[a]*Department of Computer Science, University, Country*

Abstract

Python has become the de facto platform for empirical studies in multi-objective optimization, yet most multi-objective evolutionary algorithm (MOEA) libraries remain dominated by object-oriented designs that incur substantial interpreter overhead in the inner loop. We introduce VAMOS (Vectorized Architecture for Multiobjective Optimization Studies), a research-oriented framework that represents populations as contiguous numerical arrays and factors algorithmic logic from numerical kernels. VAMOS provides interchangeable compute backends (NumPy, Numba JIT, and optional C/JAX accelerators), a unified configuration API, and nine MOEAs (NSGA-II/III, MOEA/D, SMS-EMOA, SPEA2, IBEA, SMPSO, AGE-MOEA, and RVEA). Two adaptive modules complement the core algorithms: (i) an irace-inspired racing tuner with optional multi-fidelity (Hyperband-style) warm-starting, and (ii) bandit-based adaptive operator selection for configurable operator portfolios. We also release a reproducible benchmarking pipeline that aligns problem definitions and operator settings across common Python frameworks and reports runtime and *normalized* hypervolume computed with fixed reference Pareto fronts. Across the ZDT, DTLZ, and WFG suites, VAMOS achieves up to an order-of-magnitude reduction in runtime relative to established libraries while maintaining competitive solution quality.

Keywords: Multi-objective optimization, Evolutionary algorithms,

*Corresponding author

Email address: `author@university.edu` (First Author)

Benchmarking, Hypervolume, Python framework, Vectorization

1. Introduction

Multi-objective optimization problems (MOPs) arise whenever trade-offs exist between conflicting objectives, requiring the computation of a diverse approximation of the Pareto-optimal set rather than a single optimum [1]. Multi-objective evolutionary algorithms (MOEAs) remain a dominant approach due to their robustness, anytime behavior, and ability to return sets of non-dominated solutions.

In Python, practical benchmarking of MOEAs is often constrained by two friction points. First, the ecosystem is fragmented: libraries differ in problem APIs, operator configuration, result formats, and runtime instrumentation [3, 4, 5]. Second, many implementations rely on per-solution objects and Python-level loops inside selection, variation, and survival, making the overhead of the interpreter comparable to (or larger than) the cost of typical benchmark functions. This limitation becomes acute for large-scale studies with many seeds, large populations, and repeated statistical tests.

VAMOS is designed to address both issues with a single guiding principle: *keep algorithm state in dense arrays and push hot loops into vectorized kernels.* The framework also targets reproducibility in experimental methodology: we provide a benchmark runner that standardizes problem dimensions and operators across multiple Python frameworks, and a hypervolume protocol with fixed reference Pareto fronts so that solution-quality metrics are comparable across runs and frameworks.

The contributions of this work are:

1. **Vectorized MOEA core**: a unified, array-based internal representation with pluggable compute kernels (NumPy/Numba/C/JAX) for selection, variation, and survival.
2. **Modular algorithm suite**: nine MOEAs implemented on top of shared components (termination, archives, evaluation backends, and metrics).
3. **Adaptive components**: an irace-inspired racing tuner with optional multi-fidelity warm-starting, and bandit-based adaptive operator selection (AOS) for operator portfolios.

4. **Reproducible benchmarking**: a pipeline that aligns cross-framework configurations and computes *normalized* hypervolume against fixed reference Pareto fronts distributed with the repository.

2. Related Work

2.1. Python MOEA frameworks

Several mature Python libraries exist for MOEAs. **pymoo** [3] provides a rich collection of algorithms and operators with an emphasis on modularity. **DEAP** [4] is a general evolutionary computation toolkit; while flexible, users typically assemble MOEAs manually. **jMetalPy** [5] ports the jMetal architecture to Python, offering a broad algorithm set but relying on object-centric representations. **Platypus** [6] provides a lightweight API and a set of classic MOEAs.

Table 1 summarizes high-level differences relevant to this work. The main distinction is the internal representation: VAMOS consistently uses dense arrays for populations and objectives, while most alternatives operate on per-solution objects. This difference enables kernel-based acceleration and reduces Python overhead in the inner loop.

Table 1: Comparison of Python multi-objective optimization frameworks (high-level capabilities).

Framework	**MOEAs**	**Array-based core**	**Accelerated kernels**	**Auto-config**	**AOS**
pymoo [3]	8+	Partial	No	No	No
DEAP [4]	Custom	No	No	No	No
jMetalPy [5]	11+	No	No	No	Limited
Platypus [6]	8+	No	No	No	No
VAMOS	9	Yes	Numba/C/JAX	Yes	Yes

2.2. Automatic algorithm configuration

Automatic algorithm configuration (AAC) aims to select hyperparameters that maximize performance on a target distribution of instances. irace [7] popularized iterated racing with non-parametric tests to discard poorly performing configurations early. Multi-fidelity methods such as successive halving and Hyperband evaluate many configurations with small budgets before promoting a subset to larger budgets, improving sample efficiency. VAMOS combines racing with optional multi-fidelity evaluation and warm-start checkpoints to reduce redundant computation between fidelity levels.

2.3. Adaptive operator selection

Adaptive operator selection (AOS) dynamically allocates sampling probability across operators based on online feedback [9]. Bandit-based approaches are attractive because they explicitly manage the exploration–exploitation trade-off and can be implemented with minimal assumptions. VAMOS provides a portfolio-driven AOS interface and integrates bandit policies into NSGA-II, using reward signals derived from survival and non-dominated insertions.

3. VAMOS Framework

3.1. Architecture and data model

VAMOS is organized into four layers:

1. **Foundation**: problem definitions, kernels, metrics, and archives;
2. **Engine**: algorithm implementations and shared components;
3. **Adaptation**: tuning and AOS modules;
4. **Experiment**: CLI, benchmarking utilities, visualization, and reporting.

The core design choice is to represent a population as a pair of dense arrays: decision variables $X \in \mathbb{R}^{N\times n}$ and objective values $F \in \mathbb{R}^{N\times m}$ for population size N, n decision variables, and m objectives. Variation and survival operate on these arrays (or views thereof), enabling vectorized operations (NumPy) and compilation of hot loops (Numba) without per-individual Python objects.

3.2. Compute kernels

Algorithmic logic (e.g., *NSGA-II selects survivors by non-dominated sorting and crowding distance*) is separated from numerical kernels that implement the corresponding computations. VAMOS provides multiple backends:

- **NumPy**: baseline backend using vectorized array operations.
- **Numba**: JIT compilation of hot operators and utilities, reducing Python overhead [10].

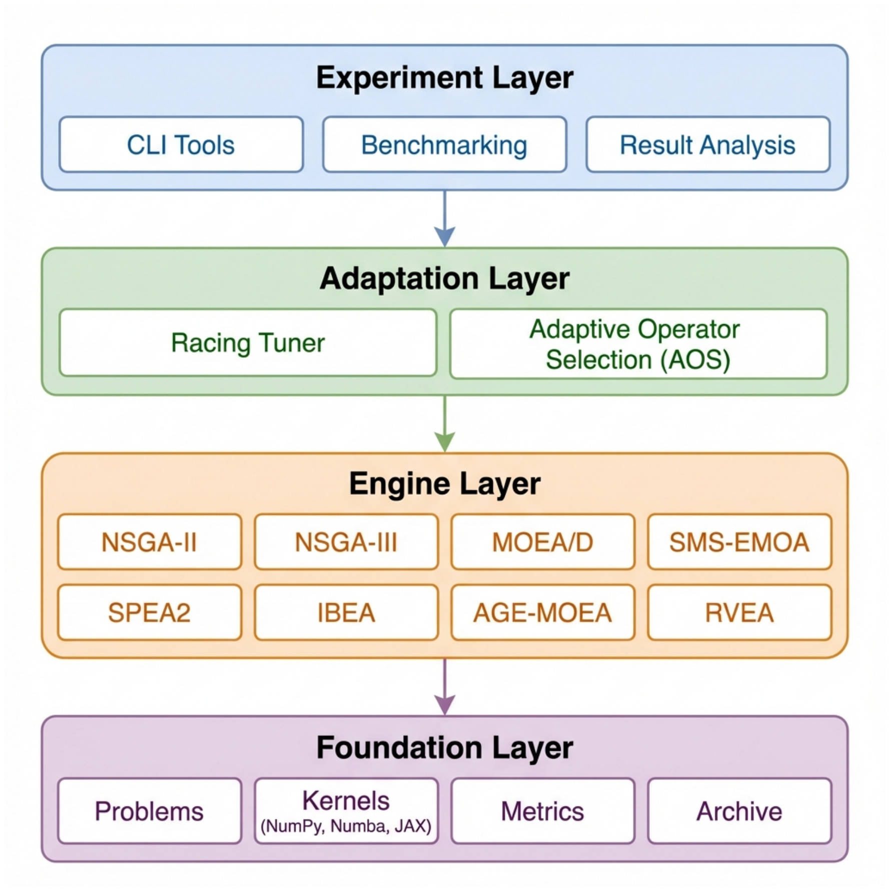

Figure 1: VAMOS four-layer architecture.

- **C-backed indicators**: optional acceleration for performance indicators such as hypervolume (e.g., via MooCore) [12].
- **JAX**: optional backend for parts of the pipeline and autodiff-based constraints, enabling GPU/TPU acceleration when available [11].

Backends are selected through configuration (e.g., `engine="numba"`) and are transparent to algorithm code.

In addition to accelerating arithmetic kernels, VAMOS reduces overhead by minimizing Python-side allocations: variation operators can reuse preallocated workspaces (e.g., scratch buffers for SBX/PM) and algorithms maintain state as contiguous arrays. This design is particularly beneficial in inner-loop routines such as tournament selection, non-dominated sorting, crowding-distance computation, and archive updates.

3.3. Algorithm suite

VAMOS implements nine multi-objective algorithms covering dominance-based, decomposition-based, indicator-based, and reference-vector paradigms. Table 2 summarizes the suite.

Table 2: Multi-objective algorithms implemented in VAMOS.

Algorithm	**Category**	**Reference**
NSGA-II	Dominance-based	[2]
NSGA-III	Reference-point	[14]
MOEA/D	Decomposition	[13]
SMS-EMOA	Indicator-based	[15]
SPEA2	Archive-based	[16]
IBEA	Indicator-based	[17]
SMPSO	Swarm-based	[18]
AGE-MOEA	Adaptive geometry	[19]
RVEA	Reference vectors	[20]

3.4. Execution model and evaluation backends

Algorithms in VAMOS expose a uniform `run(problem, termination, seed, eval_backend, live_viz)` interface, and several algorithms also support an *ask–tell* loop for streaming or interactive use. Objective evaluations are dispatched through a small evaluation-backend protocol, enabling serial evaluation for controlled benchmarking and parallel evaluation (e.g., multiprocessing or Dask-based) when objective functions are expensive. Separating evaluation from algorithmic state keeps the algorithm core deterministic under a fixed random seed and reduces the risk of framework-specific parallelism becoming a confounder in empirical comparisons.

Algorithm 1 Vectorized NSGA-II execution in VAMOS (high-level).

```
1: Sample initial population $X \in \mathbb{R}^{N \times n}$ within bounds
2: Evaluate objectives $F \leftarrow f(X)$
3: while termination criterion not met do
4:     Select mating indices $P$ via tournament on (rank, crowding)
5:     Generate offspring $X' \leftarrow \mathcal{V}(X_P)$ (crossover, mutation, repair)
6:     Evaluate offspring $F' \leftarrow f(X')$
7:     Merge: $(X, F) \leftarrow ([X; X'], [F; F'])$
8:     Compute ranks and crowding distances
9:     Select survivors of size $N$ (rank–crowding)
10: end while
11: return final non-dominated set
```

3.5. Variation pipeline and encodings

VAMOS supports real-valued, binary, and permutation encodings through a variation pipeline that composes selection, crossover, mutation, and optional repair operators. For continuous domains, the default configuration uses simulated binary crossover (SBX) and polynomial mutation (PM), with mutation probability typically set to $1/n$ where n is the number of decision variables. Encodings are normalized internally to ensure consistent operator resolution and avoid silent configuration mismatches.

3.6. Archives, metrics, and analysis tooling

The framework includes optional external archives (with configurable pruning strategies) and common indicators. Hypervolume computation can use optimized backends when installed, but VAMOS also provides fallback implementations for low-dimensional cases. Results are returned in a unified result object that supports plotting and post-hoc analysis (e.g., filtering non-dominated solutions and exporting fronts).

3.7. Adaptive operator selection (AOS)

AOS in VAMOS is portfolio-based: users specify an operator pool (e.g., multiple SBX/PM parameterizations) and a bandit policy that selects an operator per mating event. The current NSGA-II integration supports ϵ-greedy, UCB, and EXP3 policies [9], with rewards computed from survival rate and non-dominated insertions. Additional policies (e.g., Thompson sampling) are provided in the adaptation layer and can be integrated into algorithm-specific controllers.

3.8. Racing-based hyperparameter tuning

VAMOS provides an irace-inspired racing tuner [7] for automatic algorithm configuration. Candidates are evaluated across instances and seeds; underperforming configurations are eliminated as evidence accumulates. For sample efficiency, the tuner optionally uses a multi-fidelity strategy (Hyperband-style successive halving [8]) and can pass warm-start checkpoints between fidelity levels to reuse computation.

4. Experimental Evaluation

4.1. Benchmark suite and configuration alignment

We evaluate runtime and solution quality on widely used synthetic benchmarks: ZDT1–4 and ZDT6 [21] (two objectives), DTLZ1–4 and DTLZ7 [22] (three objectives), and WFG1–9 [23] (two objectives). Problem dimensionalities are fixed to standard definitions: for example, DTLZ2/3/4 use $n = m + k - 1$ with $m = 3$ and $k = 10$ (thus $n = 12$).

To ensure comparability across frameworks, we standardize the algorithmic settings of NSGA-II: population size $N = 100$, SBX crossover probability $p_c = 0.9$ and distribution index $\eta_c = 20$, polynomial mutation distribution index $\eta_m = 20$ and mutation probability $p_m = 1/n$, tournament selection, and rank–crowding survival. For each framework we map these settings to the closest available implementation.

Crucially, we align *semantics* (not only parameter names) and enforce consistent benchmark definitions. Two examples illustrate why this matters. First, in **pymoo** the polynomial mutation operator exposes both an individual-level probability (`prob`) and a per-variable probability (`prob_var`); to match the standard "$p_m = 1/n$ per decision variable" setting used by DEAP/jMetalPy, we set `prob`=1 and `prob_var`=1/n. Second, some libraries ship benchmark problems under canonical names but with library-specific domains; for instance, Platypus' built-in ZDT base class uses $[0,1]^n$, whereas ZDT4 is defined with $x_1 \in [0,1]$ and $x_{2..n} \in [-5,5]$.

Therefore, for toolkits or APIs where problem definitions differ (notably DEAP and Platypus for ZDT4), we evaluate individuals using a shared reference implementation of the benchmark function and bounds (validated against pymoo) while still using each framework's NSGA-II implementation and operators. For WFG in Platypus we similarly wrap pymoo's WFG definitions to avoid definition drift. While full equivalence cannot be guaranteed due to framework-specific details (tie-breaking, boundary handling, duplicate elimination, etc.), these choices aim to make the comparison as fair as possible by removing confounders unrelated to algorithmic overhead and numerical kernels.

4.2. Runtime measurement

All runtimes are measured wall-clock using high-resolution timers and include algorithm overhead and objective evaluations. We report medians across independent seeds and provide per-problem and per-family summaries.

4.3. Normalized hypervolume protocol

Hypervolume (HV) measures the Lebesgue measure of the region dominated by an approximation set A with respect to a reference point r (for minimization) [24]:

$$\mathrm{HV}(A,r) = \lambda\left(\bigcup_{a\in A}[a_1,r_1]\times\cdots\times[a_m,r_m]\right). \tag{1}$$

Because HV is scale-dependent and sensitive to the choice of r, naive implementations can produce values that are not comparable across runs (e.g., if r is expanded per run) and can be misleading when dominated solutions are included. To ensure a stable and comparable quality metric, we adopt the following protocol:

1. **Non-dominated filtering**: for every framework we compute HV on the final non-dominated set only.
2. **Fixed reference Pareto fronts**: for each benchmark problem we store a dense reference Pareto front P_{ref} in `data/reference_fronts/`. For ZDT and DTLZ (except DTLZ7) these fronts are generated analytically; for DTLZ7 and WFG we generate P_{ref} by dense sampling followed by non-dominated filtering.

3. **Fixed reference point**: we set $r = \max(P_{\mathrm{ref}}) + \epsilon$ (component-wise) with a small ϵ, and we disallow run-dependent expansion.
4. **Normalization**: we report $\mathrm{HV}(A, r)/\mathrm{HV}(P_{\mathrm{ref}}, r)$, yielding a dimensionless score typically in $[0, 1]$ and reducing sensitivity to objective scaling.

For WFG problems, the reference front is necessarily an approximation; we use large Pareto-set samples and non-dominated filtering, and we increase density for particularly sensitive cases (e.g., WFG2) to avoid underestimating $\mathrm{HV}(P_{\mathrm{ref}}, r)$ and obtaining normalized HV slightly above 1.

4.4. VAMOS backend comparison

Table 3 reports median runtime for VAMOS backends aggregated by problem family.

Table 3: VAMOS backend comparison: median runtime (seconds) by problem family.

Backend	**ZDT**	**DTLZ**	**WFG**	**Average**
Numba	2.57	**1.68**	**2.21**	**2.16**
Moocore	**2.31**	2.57	2.95	2.61
Numpy	5.61	5.96	7.35	6.31

4.5. Cross-framework comparison

Table 4 compares VAMOS against other Python frameworks on runtime. VAMOS is evaluated with its accelerated backend (Numba) for the headline comparison, while additional backends are reported in the appendix.

Table 4: Median runtime (seconds) by problem family across all frameworks.

Framework	**ZDT**	**DTLZ**	**WFG**	**Average**
VAMOS	**2.57**	**1.68**	**2.21**	**2.16**
pymoo	8.50	7.76	9.20	8.49
DEAP	28.00	34.02	75.80	45.94
jMetalPy	24.41	27.15	75.45	42.34
Platypus	34.35	41.21	82.28	52.61

4.6. Statistical analysis

We use the Wilcoxon signed-rank test [25] at significance level $\alpha = 0.05$ to compare paired performance distributions across seeds, per problem. Runtime is treated as a *minimization* metric, and normalized hypervolume as a *maximization* metric.

Table 5: Runtime comparison with Wilcoxon signed-rank test results.

Problem	**VAMOS (numba)**	**pymoo**	**p-value**	**Sig.**
zdt1	**2.57**	8.50	1.000	
zdt2	**2.46**	8.67	1.000	
zdt3	**3.41**	8.18	1.000	
zdt4	**1.26**	6.50	1.000	
zdt6	**2.88**	12.28	1.000	
dtlz1	**2.52**	6.88	1.000	
dtlz2	**1.33**	8.63	1.000	
dtlz3	**2.10**	7.76	1.000	
dtlz4	**1.35**	6.93	1.000	
dtlz7	**1.68**	9.20	1.000	
wfg1	**1.92**	7.54	1.000	
wfg2	**1.86**	9.20	1.000	
wfg3	**2.06**	9.11	1.000	
wfg4	**2.21**	10.14	1.000	
wfg5	**2.40**	9.65	1.000	
wfg6	**2.48**	10.05	1.000	
wfg7	**1.86**	8.70	1.000	
wfg8	**2.29**	8.93	1.000	
wfg9	**6.62**	12.87	1.000	

4.7. Discussion

The observed speedups are primarily due to (i) the array-based representation that removes per-solution Python overhead, and (ii) JIT compilation of hot operators and utilities in the Numba backend. The normalized hypervolume protocol ensures that performance gains are not obtained at the expense of solution quality and avoids artifacts caused by run-dependent reference points.

Table 6: Normalized hypervolume comparison with Wilcoxon signed-rank test results.

Problem	**VAMOS (numba)**	**pymoo**	**p-value**	**Sig.**
zdt1	0.99	**0.99**	1.000	
zdt2	0.98	**0.98**	1.000	
zdt3	0.99	**1.00**	1.000	
zdt4	0.99	**0.99**	1.000	
zdt6	**0.98**	0.98	1.000	
dtlz1	0.50	**0.91**	1.000	
dtlz2	0.75	**0.79**	1.000	
dtlz3	0.00	**0.00**	nan	
dtlz4	0.78	**0.79**	1.000	
dtlz7	0.81	**0.92**	1.000	
wfg1	0.38	**0.46**	1.000	
wfg2	**0.99**	0.94	1.000	
wfg3	**0.98**	0.89	1.000	
wfg4	0.96	**0.96**	1.000	
wfg5	0.83	**0.83**	1.000	
wfg6	**0.83**	0.82	1.000	
wfg7	0.97	**0.97**	1.000	
wfg8	**1.40**	1.37	1.000	
wfg9	**0.97**	0.94	1.000	

4.8. Threats to validity

Despite careful alignment of dimensions, budgets, problem definitions, and operator settings, cross-framework comparisons remain subject to several threats: (i) operator implementations may differ in subtle details (e.g., boundary handling, duplicate elimination, tie-breaking in survival), (ii) random number generation and seeding semantics vary across libraries, and (iii) library-specific overheads (data conversion, object materialization) may affect measured runtime. We mitigate major sources of unfairness by enforcing consistent benchmark bounds (e.g., ZDT4) and matching operator semantics where APIs differ (e.g., mutation probability in pymoo).

For hypervolume, reference fronts for problems without closed-form Pareto fronts are necessarily approximations; while we generate them densely and fix them across runs, remaining discrepancies can affect absolute normalized values, especially on WFG problems. We therefore treat normalized HV primarily as a comparative metric under a fixed protocol, and we report distributions across multiple seeds rather than relying on single-run outcomes.

4.9. Reproducibility

All experiments in this paper are reproducible from the repository. Benchmarks are executed via `python -m paper.run_paper_benchmark` (with `VAMOS_N_SEEDS` and `VAMOS_N_JOBS` controlling the number of seeds and parallel workers), and LaTeX tables are regenerated by `python paper/update_paper_tables_from_csv.py` (runtime tables) and `python paper/run_statistical_tests.py` (Wilcoxon tables). Reference Pareto fronts used for normalized hypervolume are stored in `data/reference_fronts/` and can be regenerated with the provided scripts under `experiments/scripts/` (front density is configurable via environment variables, e.g., `VAMOS_REF_WFG2_POINTS`).

4.10. LLM-assisted development workflow

LLMs were used to propose patches and accelerate mechanical refactors, while correctness and architectural integrity were enforced via automated verification. Concretely, we adopted a propose–apply–verify loop: the assistant proposes a minimal diff, the patch is applied explicitly, and we run fast-fail checks (e.g., architecture/health gates and static checks) followed by the project test/build suite and targeted benchmark smoke tests. Only patches that pass these checks are retained. This process constrains AI assistance to a productivity role and keeps the implementation and results grounded in deterministic, reproducible tooling.

5. Conclusions and Future Work

We presented VAMOS, a high-performance Python framework for multi-objective optimization studies. By combining a vectorized core with pluggable compute kernels, VAMOS reduces inner-loop overhead and enables scalable benchmarking. The framework also includes adaptive modules for automatic configuration and operator selection, and a reproducible experimental pipeline with fixed reference Pareto fronts for normalized hypervolume reporting.

5.1. Future Work

- **Many-objective benchmarking**: extend the experimental protocol beyond $m = 3$ with scalable indicators.
- **Deeper GPU integration**: expand JAX-based kernels and enable GPU-friendly operators.
- **Richer AOS rewards**: incorporate indicator deltas (e.g., HV improvements) in a principled credit assignment scheme.

Appendix A. Reference fronts and detailed benchmark results

The repository includes dense reference Pareto fronts for all benchmark problems used for normalized hypervolume computation (`data/reference_fronts/`). These fronts are generated analytically when closed forms are available (ZDT and most DTLZ problems) and by dense sampling followed by non-dominated filtering for cases where the Pareto front is disconnected or defined implicitly (DTLZ7 and WFG). For WFG2 we use a higher sampling density (configurable via `VAMOS_REF_WFG2_POINTS`) to stabilize HV normalization.

Tables A.7 and A.8 report per-problem median runtimes.

References

[1] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems, Springer, 2007.

[2] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multi-objective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.

Table A.7: Detailed VAMOS backend comparison: median runtime (seconds) per problem.

Problem	**Numba**	**Moocore**	**Numpy**
dtlz1	**2.52**	2.57	5.44
dtlz2	**1.33**	1.82	5.96
dtlz3	**2.10**	2.95	6.03
dtlz4	**1.35**	2.10	7.23
dtlz7	**1.68**	2.63	5.60
wfg1	**1.92**	2.95	6.93
wfg2	**1.86**	1.88	7.35
wfg3	**2.06**	2.79	6.80
wfg4	2.21	**1.51**	6.86
wfg5	**2.40**	3.04	6.52
wfg6	**2.48**	3.96	10.71
wfg7	1.86	**1.86**	8.50
wfg8	**2.29**	3.43	7.80
wfg9	**6.62**	6.88	8.76
zdt1	**2.57**	2.95	5.61
zdt2	2.46	**2.31**	5.39
zdt3	3.41	**2.47**	5.88
zdt4	1.26	**1.10**	6.29
zdt6	2.88	**2.14**	5.57
Average	**2.38**	2.70	6.80

Table A.8: Detailed comparison of median runtime (seconds) across all frameworks.

Problem	**VAMOS**	**pymoo**	**DEAP**	**jMetalPy**	**Platypus**
dtlz1	**2.52**	6.88	33.53	25.58	33.84
dtlz2	**1.33**	8.63	33.62	24.77	42.91
dtlz3	**2.10**	7.76	38.03	29.34	35.75
dtlz4	**1.35**	6.93	35.86	27.15	41.21
dtlz7	**1.68**	9.20	34.02	28.86	49.59
wfg1	**1.92**	7.54	47.04	46.83	74.47
wfg2	**1.86**	9.20	76.95	89.80	82.28
wfg3	**2.06**	9.11	75.80	75.45	91.78
wfg4	**2.21**	10.14	55.28	47.20	58.55
wfg5	**2.40**	9.65	53.28	45.29	63.89
wfg6	**2.48**	10.05	164.27	179.50	165.39
wfg7	**1.86**	8.70	64.21	50.15	66.53
wfg8	**2.29**	8.93	110.35	123.23	108.36
wfg9	**6.62**	12.87	257.90	308.90	267.49
zdt1	**2.57**	8.50	27.53	25.81	35.65
zdt2	**2.46**	8.67	28.00	25.73	34.35
zdt3	**3.41**	8.18	29.24	23.89	38.43
zdt4	**1.26**	6.50	29.65	24.41	30.03
zdt6	**2.88**	12.28	24.27	22.69	31.52
Average	**2.38**	8.93	64.15	64.45	71.16

[3] J. Blank, K. Deb, pymoo: Multi-objective optimization in Python, IEEE Access 8 (2020) 89497–89509.

[4] F.-A. Fortin, et al., DEAP: Evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (2012) 2171–2175.

[5] A. Benítez-Hidalgo, et al., jMetalPy: A Python framework for multi-objective optimization, Swarm Evol. Comput. 51 (2019) 100598.

[6] D. Hadka, Platypus: a free and open source Python library for multiobjective optimization, `https://github.com/Project-Platypus/Platypus`, accessed 2026.

[7] M. López-Ibáñez, et al., The irace package: Iterated racing for automatic algorithm configuration, Oper. Res. Perspect. 3 (2016) 43–58.

[8] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: A novel bandit-based approach to hyperparameter optimization, J. Mach. Learn. Res. 18 (185) (2017) 1–52.

[9] Á. Fialho, M. Schoenauer, M. Sebag, Analyzing bandit-based adaptive operator selection mechanisms, Ann. Math. Artif. Intell. 60 (2010) 25–64.

[10] S. K. Lam, et al., Numba: A LLVM-based Python JIT compiler, in: LLVM-HPC, 2015.

[11] J. Bradbury, et al., JAX: composable transformations of Python+NumPy programs, `https://github.com/google/jax`, 2018.

[12] moocore: multi-objective optimization core library, `https://github.com/multi-objective/moocore`, accessed 2026.

[13] Q. Zhang, H. Li, MOEA/D: A multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comput. 11 (6) (2007) 712–731.

[14] K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting, IEEE Trans. Evol. Comput. 18 (4) (2014) 577–601.

[15] N. Beume, B. Naujoks, M. Emmerich, SMS-EMOA: Multiobjective selection based on dominated hypervolume, Eur. J. Oper. Res. 181 (3) (2007) 1653–1669.

[16] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the strength Pareto evolutionary algorithm, TIK-Report 103, ETH Zurich, 2001.

[17] E. Zitzler, S. Künzli, Indicator-based selection in multiobjective search, in: PPSN, Springer, 2004, pp. 832–842.

[18] A. J. Nebro, J. J. Durillo, C. A. C. Coello Coello, SMPSO: A new PSO-based metaheuristic for multi-objective optimization, in: IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making, 2009, pp. 66–73.

[19] A. Panichella, An adaptive evolutionary algorithm based on non-Euclidean geometry for many-objective optimization, in: GECCO, ACM, 2019, pp. 595–603.

[20] R. Cheng, et al., A reference vector guided evolutionary algorithm for many-objective optimization, IEEE Trans. Evol. Comput. 20 (5) (2016) 773–791.

[21] E. Zitzler, K. Deb, L. Thiele, Comparison of multiobjective evolutionary algorithms: Empirical results, Evol. Comput. 8 (2) (2000) 173–195.

[22] K. Deb, L. Thiele, M. Laumanns, E. Zitzler, Scalable multi-objective optimization test problems, in: Congress on Evolutionary Computation (CEC), 2002, pp. 825–830.

[23] S. Huband, P. Hingston, L. Barone, L. While, A review of multiobjective test problems and a scalable test problem toolkit, IEEE Trans. Evol. Comput. 10 (5) (2006) 477–506.

[24] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, V. G. da Fonseca, Performance assessment of multiobjective optimizers: An analysis and review, IEEE Trans. Evol. Comput. 7 (2) (2003) 117–132.

[25] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics Bull. 1 (6) (1945) 80–83.