HumpDay: Derivative-Free Optimizers in Pure Python and JavaScript, with a Contamination-Resistant Real-World Benchmark

Peter Cotton
Microprediction
peter.cotton@microprediction.com
https://humpday.microprediction.org

July 6, 2026

Abstract. The Python package **humpday** provides twenty-three derivative-free optimizers behind one uniform contract: minimise a black-box function on the unit cube under a hard evaluation budget. Every algorithm is implemented in pure Python with no required dependencies (a **numpy** backend is used transparently when present) and again in zero-dependency JavaScript, with parity tests holding the two in agreement, so the same optimizer runs on a server or in a browser. A recommender selects an algorithm from the problem's dimension, budget, and measured evaluation cost, using rankings precomputed on the package's own benchmark: eighty-one objectives ported from real applications, each paired with an interactive browser demonstration. Scoring applies seeded smooth bijections of the cube that relocate every optimum, so no method, human or machine, can score by memorising solutions; the suite remains valid when the candidates are written by language models. We describe the design and two findings the benchmark has produced: optimizer rankings on synthetic test functions correlate only moderately with rankings on the real-world suite (Kendall τ between 0.42 and 0.49 across budgets), and the suite supported the generation and out-of-sample validation of Alloy, a machine-designed optimizer that now ships in the package.

Keywords: derivative-free optimization, black-box optimization, benchmarking, algorithm selection, benchmark contamination, Python, JavaScript, **humpday**.

1 Introduction

A practitioner with an expensive black-box objective and a budget of a few hundred evaluations faces two problems. The first is having good optimizers available without a compilation step or a dependency tree. The second is knowing which one to use, because the answer changes with dimension, budget, and the character of the objective.

Excellent algorithm collections exist: **scipy** [Virtanen et al., 2020], **nevergrad** [Rapin and Teytaud, 2018], **NLopt** [Johnson, 2007], **pymoo** [Blank and Deb, 2020], and the PRIMA re-implementations of Powell's solvers [Ragonneau and Zhang, 2024]. Benchmark platforms also exist, notably COCO/BBOB [Hansen et al., 2021] and IOHprofiler [Doerr et al., 2018]. **humpday** began as a package that wrapped and ranked the collections; the version described here inverts that design. Every algorithm is now a first-class pure-Python port with no required dependencies, twinned in JavaScript, and the rankings that drive its recommendations come from its own benchmark of real-application objectives rather than from synthetic test functions.

Two properties distinguish the package from its neighbours. The first is the dual implementation: parity tests hold the Python and JavaScript implementations in agreement, so every algorithm, and every benchmark problem, runs identically in a browser, which is

where the package's documentation lives. The second is contamination resistance. Benchmark objectives derive from public material, and modern candidates may be written by language models trained on that material; **humpday** scores every run through a seeded smooth bijection of the cube that relocates the optimum, so memorised solutions earn nothing. Section 5 shows this is not a hypothetical concern: the suite has already been used to generate, select, and validate an optimizer written entirely by a language model.

Section 2 shows the package in use. Section 3 describes the design. Section 4 describes the benchmark and its disguise mechanism. Section 5 reports the two findings, and Section 6 closes with scope and limitations.

2 Using the package

Installation is `pip install humpday`, with no required dependencies. The everyday interface is a single call:

```
from humpday import minimize

def objective(x):
    return (x[0] - 0.3) ** 2 + abs(x[1] - 0.6)

result = minimize(objective, bounds=[(-1, 1), (0, 2)],
                  options={"n_trials": 200})
```

Rectangular bounds are mapped to the unit cube internally. When `method` is omitted, the algorithm is chosen by the recommender (Section 3) from the dimension, the budget, and the measured cost of one objective call. Passing `method="Alloy"` or any other roster name overrides the choice.

The lower-level contract is deliberately plain. Every optimizer is a class with the signature

```
optimizer = Alloy(objective, n_trials, n_dim)
best_value, best_x = optimizer.optimize()
```

where `objective` takes a list of `n_dim` floats in $[0, 1]$ and the budget `n_trials` is a hard cap on objective calls. An ask/tell interface drives any optimizer incrementally when the caller owns the evaluation loop, and produces trajectories identical to the monolithic run.

The same roster is available in JavaScript:

```
const { Alloy } = require('humpday');
const opt = new Alloy(objective, 200, 2);
const { bestValue, bestX } = opt.optimize();
```

3 Design

3.1 One contract, two languages

The twenty-three optimizers span six classical families (Powell-style trust region, simplex and direct search, evolutionary and swarm methods, annealing and other metaheuristics, model-based sampling, and pure baselines) plus one machine-designed member (Section 5). All operate on $[0, 1]^n$ under a hard evaluation budget, and nothing else: no gradients, no

Jacobians, no constraint callbacks. Problems with other domains reach the cube through transforms shipped with the package, including an invertible map between the cube and the probability simplex, so portfolio and allocation problems with weights summing to one are optimized by unmodified box-domain algorithms.

The Python implementations read their array primitives from a small shim that re-exports **numpy** when it is installed and a pure list backend otherwise; algorithm code is identical either way. The JavaScript implementations are ports of the same logic with no dependencies at all. A parity suite runs both on identical problems and fails if either implementation regresses, which keeps the browser demonstrations evidential rather than decorative: what runs on the documentation site is the algorithm being documented.

3.2 The recommender

Algorithm choice is data. Each optimizer carries a cost tier, and a precomputed grid stores, for a lattice of (dimension, budget) cells, the Borda mean rank of every algorithm across the benchmark suite. Given a problem, `humpday.eligibility.recommend` filters by tier against the measured cost of one objective call, then returns the eligible algorithm with the best mean rank on the nearest cell. An opt-in cost-aware variant penalises algorithms whose own overhead would dominate a cheap objective. The recommendation is therefore reproducible arithmetic on published benchmark results, not editorial opinion.

4 The benchmark

The suite contains eighty-one objectives ported from real applications: enzyme kinetics, wind-farm layout, battery dispatch, lens design, antenna arrays, pool break shots, rocket landing, and others, spanning 2 to 100 dimensions. Each ships in both languages, and each has an interactive browser page where every optimizer in the roster can be raced on the actual problem.

Raw objectives of this kind have a weakness as benchmarks: they are built from public material, so their solutions may be memorised, by people, by tuned heuristics, and now by language models that write candidate code. **humpday** therefore never scores a raw objective. A seeded smooth bijection of the cube (a composition of coordinate warps) is applied inside the objective, relocating the optimum while preserving the problem's character; each seed yields a different disguised instance. A candidate that knows where the answer used to live gains nothing, and a suite of disguised instances measures search, not recall.

Scores are reported as regret normalised against a fixed panel of standard optimizers run on the same instance: 0 means matching the best panel member everywhere, 1 the worst. This makes results comparable across objectives whose raw scales differ by orders of magnitude.

5 Two findings

5.1 Synthetic rankings transfer only moderately

Do rankings earned on synthetic test functions predict rankings on real problems? The package's full roster, plus a reference CMA-ES from **nevergrad**, was ranked on a battery of synthetic functions of the BBOB style, and independently on the disguised real-world suite, at three budgets. Table 1 reports the rank correlations.

budget	Kendall τ	p	Spearman ρ
60	0.49	0.001	0.66
120	0.49	0.001	0.67
240	0.42	0.006	0.58

Table 1: Rank correlation between the 22-optimizer leaderboard on synthetic test functions and on the disguised real-world suite, per evaluation budget.

The correlation is positive and significant, and far from 1. Roughly, a synthetic leaderboard explains under half the ordering that matters on the real suite, and the disagreement grows with budget. Trust-region methods that dominate smooth synthetic functions drop sharply on the real problems, while robust direct-search methods rise. This is the empirical case for recommending from real-application rankings, and a caution against transplanting synthetic benchmark conclusions into practice.

5.2 The suite supports algorithm discovery: Alloy

Because the disguise mechanism makes scores meaningful even when candidates are machine-written, the suite can drive search over algorithms themselves. In a companion note [Cotton, 2026], a language model was prompted to blend the mechanisms of five classical optimizers in stated proportions; candidates were selected on disguised instances and the winner was validated on twenty-nine objectives never used in any selection step. That winner, Alloy, had the best mean rank at every budget from 60 to 480 evaluations against six competitors including CMA-ES, winning 64% to 77% of 580 instances pairwise (sign-test $p < 10^{-10}$). Alloy now ships in the package as an ordinary roster member, in both languages.

The companion note reports the failures alongside: most blends are mediocre, single generations vary severalfold, and a tuned generic template matched every in-sample score yet collapsed out of sample. The benchmark's out-of-sample discipline, problems verified untouched against the history of the suite, is what separated the finding from the noise.

6 Discussion

humpday's scope is deliberately narrow: box-bounded, derivative-free, serial, small-budget optimization. Users with gradients, constraints beyond bounds and the simplex, or budgets of 10^5 evaluations are better served elsewhere, and the recommender will not pretend otherwise. Within scope, the package offers a property the larger ecosystems do not: the algorithm you read about, the algorithm you run in the browser demonstration, and the algorithm you import are verifiably the same object, ranked on problems that resist memorisation.

The benchmark's disguise mechanism addresses training-set contamination for solution locations, not for problem structure: a model that has seen a problem family will still recognise its shape. And the suite's problems, however varied, are one curation; the rank-correlation finding cuts both ways, cautioning against over-trusting any single suite, including this one.

7 Availability

humpday is MIT-licensed at https://github.com/microprediction/humpday, installable with `pip install humpday`, with the JavaScript roster published as the **humpday** package on npm. Documentation, the interactive demonstrations, and the working papers behind Section 5 are at https://humpday.microprediction.org. Every number in this paper is reproducible from run logs in the repository's `papers/` directory.

References

Julian Blank and Kalyanmoy Deb. pymoo: Multi-objective optimization in Python. *IEEE Access*, 8:89497–89509, 2020.

Peter Cotton. Alloy: A machine-designed derivative-free optimizer, and a full account of the search that found it, 2026. Working paper, https://humpday.microprediction.org/papers.html.

Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. In *arXiv:1810.05281*, 2018.

Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36(1):114–144, 2021.

Steven G. Johnson. The NLopt nonlinear-optimization package, 2007. https://github.com/stevengj/nlopt.

Tom M. Ragonneau and Zaikun Zhang. PDFO: A cross-platform package for Powell's derivative-free optimization solvers. *Mathematical Programming Computation*, 2024. See also the PRIMA project, https://www.libprima.net.

Jérémy Rapin and Olivier Teytaud. Nevergrad – a gradient-free optimization platform, 2018. https://github.com/facebookresearch/nevergrad.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272, 2020.