ImprovEvolve: Basin-Hopping Meets LLM-Guided Evolutionary Search

Alexey Kravatskiy [1] **Valentin Khrulkov** [2 3] **Ivan Oseledets** [4 3]

Abstract

LLM-guided evolutionary computation, most notably *AlphaEvolve*, has been remarkably successful in discovering novel mathematical constructions by solving challenging optimization problems. The standard approach is to evolve a monolithic program that directly outputs a candidate solution. We present **ImprovEvolve**, an algorithmic alternative that drastically reduces cognitive load on the LLM. Instead of prompting the model for an end-to-end optimizer, we evolve a program with three specialized operators of initialization, local improvement, and perturbation. We then approach the optimum by iteratively applying local improvements and intensity-scheduled perturbations, effectively driving a basin-hopping search with LLM-evolved subroutines. For hexagon in hexagon packing, *ImprovEvolve* discovers new state-of-the-art packings of 11, 12, 15, and 16 hexagons, and additionally for 14, 17, and 23 hexagons after minimal expert tuning of the generated code. For the second autocorrelation inequality, the evolved and human-scaled program pushes the lower bound from 0.96102 to 0.96258. For spherical codes, the *ImprovEvolve* program lowers the best-known maximum cosine for the majority of 90 randomly chosen diverse state-of-the-art spherical codes, achieving relative improvements of up to 2.4%.

1. Introduction

AlphaEvolve (Novikov et al., 2025) combines powerful Large Language Models (LLMs) with the classical evolutionary algorithm MAP-Elites (Mouret & Clune, 2015) to solve mathematical optimization problems. Given a task, the user specifies a validation function and a set of seed programs; candidate solutions are then iteratively refined by selecting promising parents from the archive and recombining them with an LLM-based mutation operator. The original implementation has not been released, but several open-source reimplementations are available (Sharma, 2025; Lange et al., 2025; Assumpção et al., 2026; Khrulkov et al., 2025).

This work focuses on the ability of LLMs to generate optimization code for problems with extremely rugged landscapes, whose optima often exhibit intricate structure, a phenomenon familiar from packing problems. Designing an effective optimizer *ab initio* is difficult for an LLM: it must produce an end-to-end procedure together with a non-trivial initialization scheme. In practice, generated programs lean on off-the-shelf routines such as `L-BFGS-B`, which are inadequate for the non-convex, non-smooth landscapes that arise once all constraints are enforced. As a result, LLM-generated programs blend mathematical insight with optimization heuristics rather than rigorously solve the optimization problem.

When such a program yields only an imperfect solution, or when a non-trivial solution has already been constructed by domain experts, local search can usually improve it. This motivates **ImprovEvolve**: instead of evolving a program that directly outputs an optimal solution (as in *AlphaEvolve*), we evolve a class with the following interface:

- `improve`(x): returns x' with fitness at least that of x;
- `perturb`(x, σ): randomly explores the search space at intensity σ;
- `generate_config`$(seed)$: returns a feasible initial solution from a given random seed.

We then approach the optimum as follows: sample initial configurations with `generate_config`, refine each via `improve` and select the best, then iteratively apply `perturb` and `improve` with a scheduled intensity, greedily accepting only improvements (Figure 1).

This decomposition offers three advantages. First, it reduces the LLM's cognitive load by separating initialization, local improvement, and exploration into distinct concerns, each with a natural slot for domain knowledge. Second, the modular structure makes evolved programs easier for both

[1]MIRIAI, Russia [2]FusionBrain Lab, Russia [3]AXXX, Russia [4]Institute of Numerical Mathematics, Russia. Correspondence to: Alexey Kravatskiy <kravatskii.a@miriai.org>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026. Copyright 2026 by the author(s).

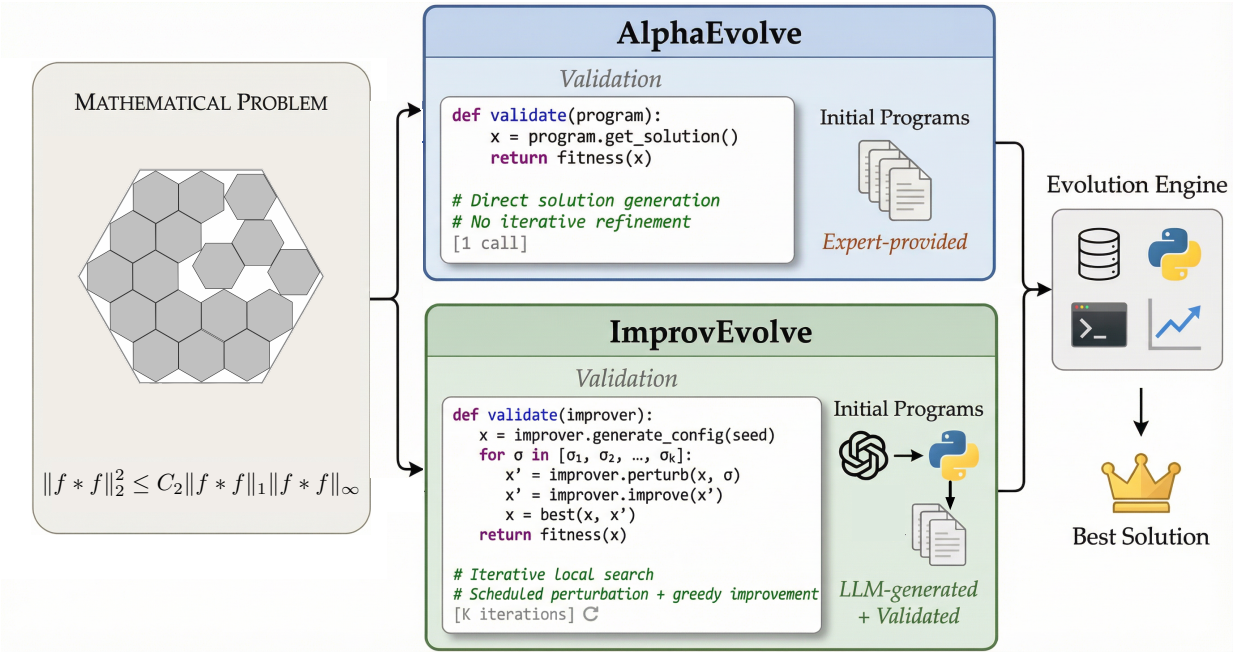


Figure 1. **Comparison of *AlphaEvolve* and *ImprovEvolve*.** *Top:* In *AlphaEvolve*, the LLM evolves a program that directly outputs a candidate solution, requiring the model to design an end-to-end optimization algorithm including initialization, search strategy, and termination criteria. *Bottom:* In *ImprovEvolve*, the LLM evolves a class with three modular methods: `generate_config` (initialization), `improve` (local optimization), and `perturb` (exploration), which are combined via basin-hopping dynamics with a scheduled perturbation intensity. This decomposition reduces the cognitive burden on the LLM by separating distinct optimization concerns into tractable subproblems. The pseudocode shown for *ImprovEvolve* is simplified; see Algorithm 1 for the full description.

humans and LLMs to interpret and debug. Third and most important, the evolved class can be applied to *any* starting solution, such as one provided by a domain expert and representing a sophisticated construction beyond the LLM's immediate grasp.

2. Related Work

Evolutionary search has a long history in machine learning. Most directly relevant to this work, Romera-Paredes et al. (2024); Novikov et al. (2025) introduced *FunSearch* and *AlphaEvolve* to evolve programs that solve mathematical problems, with mutations guided by powerful LLMs such as Gemini 2.5 Pro (Comanici et al., 2025). *FunSearch* evolves a single Python function that constructs a solution; *AlphaEvolve* generalises this to arbitrary mutable code blocks with richer evolutionary feedback. Georgiev et al. (2025) subsequently evaluated *AlphaEvolve* at scale on cutting-edge mathematical challenges.

AlphaEvolve has inspired numerous open-source implementations and extensions (Sharma, 2025; Lange et al., 2025; Khrulkov et al., 2025; Assumpção et al., 2026; Wang et al., 2025; Yan et al., 2026; Yuksekgonul et al., 2026). Most of these refine the evolutionary loop or replace it altogether, e.g., via reinforcement learning (Wang et al., 2025; Yuksekgonul et al., 2026). We instead address a more fundamental obstacle: the difficulty of asking an LLM to generate effective optimization code at all. We do so by reformulating the program the LLM is asked to produce; to our knowledge, this is the first work to apply such a reformulation in the context of LLM-guided evolution. As the backbone of *ImprovEvolve*, we adopt the open-source *GigaEvo* framework[1] of Khrulkov et al. (2025), chosen for its flexibility and its comprehensive collection of pre-implemented mathematical problems.

A complementary line of work clarifies the difficulty of these landscapes from the classical-optimization side: Berthold et al. (2026) cast several combinatorial geometry problems (including hexagon packing) as nonlinear programs and obtained state-of-the-art results using commercial solvers such as FICO Xpress and SCIP. We tackle the same problems from the opposite direction, showing that incorporating classical global-optimization techniques, namely basin-hopping, directly into an LLM-guided evolutionary framework yields superior solutions; with lightweight human adjustments, *ImprovEvolve* matches or exceeds these formalized global-solver outputs.

[1] https://github.com/FusionBrainLab/gigaevo-core

3. Methodology

Our approach reformulates what the LLM is asked to produce so that classical global-optimization machinery does the heavy lifting on the search side, leaving domain-specific structure to the evolved code.

Connection to Basin-Hopping. Basin-hopping (Wales & Doye, 1997) is a well-known global-optimization technique, particularly effective for high-dimensional landscapes with many local minima separated by large barriers. The algorithm iterates (1) a random perturbation of the current solution, (2) local optimization from the perturbed point, and (3) Metropolis acceptance at temperature T; at $T = 0$ it reduces to *Monotonic Basin-Hopping*, accepting only improvements.

ImprovEvolve can be viewed as a variant of basin-hopping in which the perturbation and local-optimization subroutines are themselves *evolved* by an LLM rather than fixed generic operators: $\texttt{perturb}(x, \sigma)$ replaces the random coordinate step (with σ controlling intensity), $\texttt{improve}(x)$ replaces the local minimizer (e.g., L-BFGS), and we use $T = 0$ (monotonic acceptance) to accelerate convergence during validation. This shifts the LLM's task from designing an end-to-end optimizer to producing domain-specific `improve` and `perturb` operators, thereby setting a more tractable objective that directly leverages the LLM's ability to encode mathematical structure and heuristics.

Validation Scheme. The validation scheme for *ImprovEvolve* (Algorithm 1) proceeds in two stages. In *Stage A*, K candidate solutions are generated with distinct random seeds and immediately refined via `improve`; the highest-fitness candidate becomes the starting point $\boldsymbol{x}^*$. Because the final result depends heavily on which basin the search starts in, sampling multiple initial configurations and keeping the best of them reduces sensitivity to poor initializations. In *Stage B*, the algorithm performs R rounds of perturbation and improvement. Within each round, the perturbation intensity σ decays geometrically from $\sigma_{\max}$ to $\sigma_{\min}$ over M iterations, encouraging broad exploration early (large perturbations to escape local minima) and fine-grained refinement later (small perturbations to converge inside the current basin). Each perturbed solution is refined via `improve` and accepted only if it strictly improves the current best (monotonic acceptance, $T = 0$). Restarting the σ schedule on every round lets the search repeatedly attempt large jumps while still refining the incumbent.

Initialization Strategy. To accelerate evolution without human assistance, we use *Gemini 3 Pro* to generate $N = 5$ initial programs and iteratively refine them in the same chat to fix validator errors (e.g., out-of-bounds configurations). Since running everything with a top-tier model is prohibitively expensive (Georgiev et al., 2025), we adopt a two-tier approach: *Gemini 3 Pro* for initialization, and the faster *Gemini 3 Flash Preview* for evolution. The initial programs produced *from the same prompt* already exhibit a diverse set of approaches and are competitive out of the box, consistent with Gideoni et al. (2025).

Algorithm 1 Validation Scheme for *ImprovEvolve*

Input: Evolved class with methods `generate_config`, `improve`, `perturb`; evaluation function `fitness`; number of rounds R
Output: Solution $\boldsymbol{x}^*$ with highest fitness found
{Hyperparameters: $K \leftarrow 10$, $M \leftarrow 10$, $\sigma_{\max} \leftarrow 100$, $\sigma_{\min} \leftarrow 0.001$}
{Stage A: Initialization}
for $s = 1$ **to** K **do**
  $\boldsymbol{x}_s \leftarrow \texttt{improve}(\texttt{generate_config}(s))$
end for
$\boldsymbol{x}^* \leftarrow \arg\max_{s \in \{1,\ldots,K\}} \texttt{fitness}(\boldsymbol{x}_s)$
{Stage B: Basin-hopping with scheduled perturbation}
for round $= 1$ **to** R **do**
  for $t = 1$ **to** M **do**
    $\sigma \leftarrow \sigma_{\max} \cdot (\sigma_{\min}/\sigma_{\max})^{(t-1)/(M-1)}$
    $\boldsymbol{x}' \leftarrow \texttt{improve}(\texttt{perturb}(\boldsymbol{x}^*, \sigma))$
    if $\texttt{fitness}(\boldsymbol{x}') \geq \texttt{fitness}(\boldsymbol{x}^*)$ **then**
      $\boldsymbol{x}^* \leftarrow \boldsymbol{x}'$ {monotonic acceptance}
    end if
  end for
end for
return $\boldsymbol{x}^*$

Evolutionary setup. We build on the open-source *GigaEvo* framework (Khrulkov et al., 2025), which provides a modular evolutionary engine combining a Redis-backed candidate store, an asynchronous DAG execution engine for evaluation pipelines, a MAP-Elites loop with configurable island topologies, and an LLM-based mutation operator. We use a *single-island* MAP-Elites configuration with fitness as the sole behavior descriptor, discretized into 150 bins; this untuned resolution proved sufficient for state-of-the-art results on all benchmarks, while preliminary multi-island experiments (e.g., with code complexity as a second descriptor) offered no benefit and only added system complexity. We use *GigaEvo*'s default mutation pipeline with system prompts unchanged from the repository: `InsightsStage` (LLM-generated improvement hints), `LineageInsights` (parent–child fitness feedback), and aggregate statistics.

It is useful to distinguish two nested levels of optimization. The *outer loop* (MAP-Elites) maintains a population of candidate *programs*, each implementing the `generate_config`/`improve`/`perturb` interface.

The *inner loop* (Algorithm 1) executes a single candidate via basin-hopping to produce a *solution*, and assigns the program a fitness equal to the best solution quality found during that run. For *GigaEvo* baselines, each program is invoked in a single call to directly produce a candidate solution. At each generation, N_{elites} programs are sampled from the archive with probability proportional to fitness, and $N_{\text{offspring}}$ offspring are produced by passing N_{parents} uniformly sampled elites to the LLM, together with their metrics and the pipeline context above; each offspring is evaluated by Algorithm 1 and inserted into the archive if its fitness warrants. The number of basin-hopping rounds R controls the trade-off between evaluation cost and solution quality: we use a smaller R during evolution for rapid fitness estimation and a larger R for final validation; problem-specific values appear below.

Computational cost. The wall-clock time of a single evolutionary run is dominated by the cost of each `improve` call. For hexagon packing, where the inner optimization is a moderate-dimensional constrained problem ($3n$ variables), a full run takes roughly 10 hours on a single machine. The second autocorrelation inequality is considerably more expensive: each `improve` call performs high-dimensional L-BFGS optimization over step functions with up to tens of thousands of parameters, giving runs of roughly 40 hours.

4. Benchmark problems

We evaluate *ImprovEvolve* on three challenging optimization problems: hexagon packing, the second autocorrelation inequality, and spherical codes.

As the backbone LLM we use *Gemini 3 Pro* (via chat at `aistudio.google.com`) to generate initial programs, and *Gemini 3 Flash Preview* (via `openrouter.ai`, default settings, temperature $T = 1$) to run the evolution. Common evolution parameters are $N_{\text{elites}} = 6$, $N_{\text{parents}} = 2$, $N_{\text{offspring}} = 10$, and all programs run on CPU to allow parallelization. During evolution, a program is discarded if its `improve` function returns an invalid configuration; this requirement is relaxed during final validation, since even reliable optimizers can occasionally diverge after an unlucky perturbation over an extended run.

4.1. Hexagon packing

The hexagon packing (HEX n) problem asks to pack n unit regular hexagons (circumradius $r = 1$) inside a flat-topped regular enclosing hexagon of side length L, minimising L subject to a no-overlap constraint. Each hexagon is parameterised by its center (x_i, y_i) and rotation angle θ_i, giving a $3n$-dimensional search space. Following Novikov et al. (2025) and Georgiev et al. (2025), we evolve programs for $n = 11$—where the best previously known side length was $L = 3.9434$ (Friedman, 2025), subsequently improved to $L = 3.9301$ by *AlphaEvolve* (Novikov et al., 2025)—and validate the evolved programs across a range of n.

Table 1. Comparison of best known results for hexagon packing (HEX n) and the second autocorrelation inequality (ACI 2).

Method	HEX 11 (↓)	HEX 12 (↓)	ACI 2 (↑)
Human	3.9434 (Friedman, 2025)	4.0000	0.94136 (Jaech & Joseph, 2025)
AlphaEvolve	3.9301 (Novikov et al., 2025)	3.9419	0.96102 (Georgiev et al., 2025)
ThetaEvolve (Wang et al., 2025)	–	–	0.94690
CodeEvolve (Assumpção et al., 2026)	3.9379	4.0000	0.88110
Global Solvers (Berthold et al., 2026)	3.9249	3.9417	–
GigaEvo (baseline) (Khrulkov et al., 2025)	3.9296	invalid[1]	0.9478
ImprovEvolve	**3.9245**	**3.9416**[2]	0.9512
ImprovEvolve+E[3]	**3.9245**	**3.9416**	**0.96258**

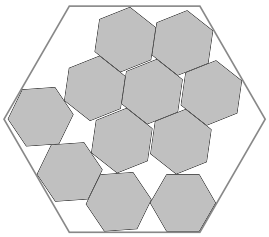
[1] Hexagon overlap encountered when validating program evolved for $n = 11$ on $n = 12$.
[2] Value obtained by validating code evolved for $n = 11$ on $n = 12$.
[3] *ImprovEvolve+E*: the evolved program was lightly edited by a human expert. For HEX, the optimizer and convergence parameters were adjusted (see Section A.2.1); for ACI 2, the program was modified to support an effective start from *AlphaEvolve*'s solution (see Section B.2.1).

Table 2. ImprovEvolve hyperparameters for HEX 11 and ACI 2 problems.

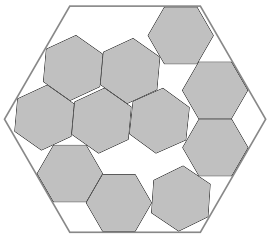
Parameter	HEX 11	ACI 2
Timeout	5 min	20 min
Init. seeds (K)	10	3
BH rounds (R)	15	5
Iter. per round (M)	11	6
σ schedule	`[1e2, 5e1, 1e1, 5, 1, 5e-1, 1e-1, 5e-2, 1e-2, 5e-3, 1e-3]`	`[1e2, 1e1, 1, 1e-1, 1e-2, 1e-3]`

The fitness is $-L$, with any containment or non-overlap violation raising an exception and discarding the configuration. The optimization landscape is highly non-convex: a small perturbation in position or rotation can introduce an overlap or a containment violation, making the problem particularly hard for evolutionary search. Hexagon packing is nevertheless an effective testbed for our approach, since perturbations are geometrically interpretable and configurations are easy to visualise.

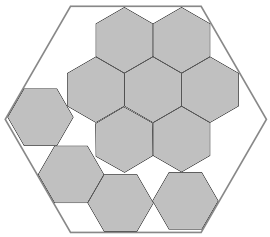
The task description supports any n from 1 to 25, but evolution was run for a single fixed value $n = 11$, using the *ImprovEvolve* hyperparameters in Table 2. The run produced two qualitatively different state-of-the-art packings: the first (Figure 2b) is structurally distinct from *AlphaEvolve*'s packing (Figure 2a) and gives a noticeably shorter enclosing side, while the second (Figure 2c) closely resembles *AlphaEvolve*'s structural basin but reaches a deeper



(a) AlphaEvolve ($L = 3.9301$)



(b) ImprovEvolve ($L = 3.9269$)



(c) ***ImprovEvolve*** **SOTA** ($L = 3.9245$)

Figure 2. Three packings of $n = 11$ unit hexagons. (a) The previous state-of-the-art by *AlphaEvolve* (Novikov et al., 2025). (b) A structurally distinct packing found by *ImprovEvolve*. (c) The new overall best packing discovered by *ImprovEvolve*, which recovers a structure similar to *AlphaEvolve* but finds a deeper optimum.

optimum. The existence of two such different high-quality packings indicates that the fitness landscape contains multiple deep basins, underscoring the importance of Stage A diversity.

We then validated the evolved program on a GPU for all $11 \leq n \leq 24$ under increased computational budgets: $K = 100$ initial seeds, $R = 100$ basin-hopping rounds, and a σ schedule of `np.geomspace(1000, 0.001, 25)`. This yielded new state-of-the-art packings for $n = 12, 15$, and 16 (extended results in Table 3 and Figure 8, both in Appendix A). Crucially, the same program evolved for $n = 11$ generalises to unseen problem sizes without retraining or re-evolution, demonstrating the robustness of the `improve`/`perturb` interface.

We also consider **ImprovEvolve+E**, a variant in which a human expert applies minor, targeted edits to the evolved program. For the hexagon task, three changes were made (annotated diff in Section A.2.1): (i) replacing the `L-BFGS-B` optimizer with `SLSQP`, which solves a quadratic subproblem at each step and is better suited to the constrained geometry, though less numerically stable; (ii) softening the initial penalty weights for a gentler warm-up; and (iii) relaxing the variable bounds and increasing the iteration limit to allow finer convergence. These edits preserve the overall algorithmic structure produced by the LLM while leveraging domain knowledge to improve robustness. They yield

Table 3. Enclosing hexagon side lengths for HEX 13–30. *ImprovEvolve+E* denotes minor human edits (see Section A.2.1). Bold digits improve on the global solver baseline. A dash (–) indicates no previously reported packing.

Task	Human (Friedman, 2025)	Solvers (Berthold et al., 2026)	*ImprovEvolve*	*ImprovEvolve+E*
HEX 13 (↓)	4.0000	–	4.0000	4.0000
HEX 14 (↓)	4.2724	4.2**690**	4.2724	4.2**690**
HEX 15 (↓)	4.4541	4.4477	4.4**473**	4.4**473**
HEX 16 (↓)	4.5363	4.5279	4.5**275**	4.5**275**
HEX 17 (↓)	4.6188	4.614+	4.6188	4.61**36**
HEX 18 & 19 (↓)	4.6188	–	4.6188	4.6188
HEX 20 & 21 (↓)	5.0000	–	5.0000	5.0000
HEX 22 (↓)	5.2856	–	5.2857	5.2856
HEX 23 (↓)	5.4286	5.413+	5.4848	5.4**000**
HEX 24 (↓)	5.4848	–	5.4848	5.4848
HEX 25 (↓)	–	–	5.6510	5.6239
HEX 26 (↓)	–	–	5.7142	5.7097
HEX 27 (↓)	–	–	5.7142	5.7142
HEX 28 (↓)	–	–	5.9723	5.9089
HEX 29 (↓)	–	–	6.0000	6.0000
HEX 30 (↓)	–	–	6.0045	6.0000

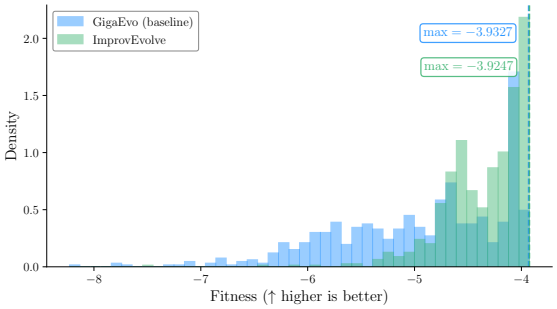


Figure 3. Distribution of validation fitness values ($-L$) across all valid programs produced during evolution for the $n = 11$ hexagon packing problem. *ImprovEvolve* consistently yields higher-fitness programs compared to the *GigaEvo* baseline under identical evolutionary parameters and time limits.

additional state-of-the-art packings for $n = 14, 17$, and 23 (Table 3, Figure 8). We further evaluated the edited program for $25 \leq n \leq 30$, where no packings have previously been reported, and obtained structured, non-chaotic packings (Figure 8).

Under identical evolutionary parameters and time limits, *ImprovEvolve* proved substantially more effective than the *GigaEvo* baseline; Figure 3 shows the distribution of validation fitness values across the run. Moreover, validating the best *GigaEvo* program for $n > 11$ produced overlap errors, indicating that its solution did not transfer to other problem sizes. When the full complexity of an optimization procedure is encoded in a single program, the LLM struggles to produce a solution that generalises reliably.

Further details, including the prompt and the evolved program, are provided in Appendix A.

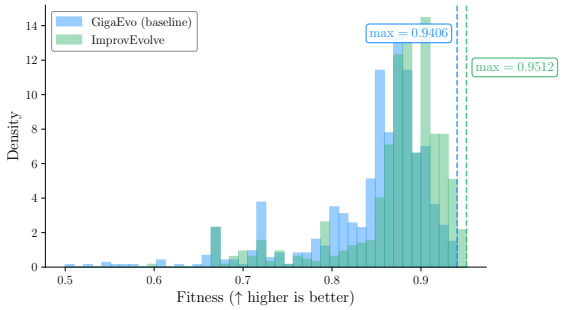
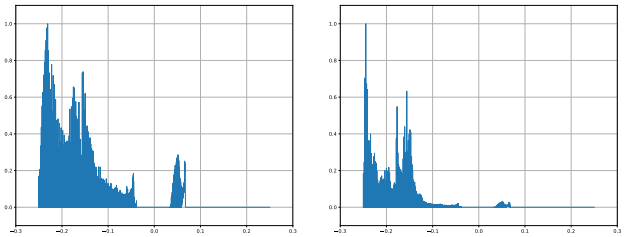


Figure 4. Distribution of validation fitness values $C(f)$ across all valid programs produced during evolution for the ACI 2 problem. *ImprovEvolve* yields consistently higher fitness than the *GigaEvo* baseline.



(a) AlphaEvolve *(b) ImprovEvolve+E*

Figure 5. Extremal functions f for the second autocorrelation inequality. (a) *AlphaEvolve*'s solution ($C(f) = 0.96102$, 50 000 steps) (Georgiev et al., 2025). (b) *ImprovEvolve+E* solution ($C(f) = 0.96258$, 1.6M steps), obtained by resuming optimization from (a). Despite the small difference in fitness, the two functions are qualitatively different in structure.

4.2. Second autocorrelation inequality

To test *ImprovEvolve* on problems with highly parameterised constructions, we consider the second autocorrelation inequality (ACI 2). For a non-negative function $f\colon \mathbb{R} \to \mathbb{R}$, define the autoconvolution

$$(f * f)(t) = \int_{\mathbb{R}} f(t-x)\, f(x)\, dx,$$

and let C_2 be the smallest constant such that

$$\|f * f\|_2^2 \;\le\; C_2\, \|f * f\|_1\, \|f * f\|_\infty$$

holds for all non-negative f. It is known that $0.88922 \le C_2 \le 1$ (Georgiev et al., 2025); the goal is to tighten the lower bound by finding f that maximises the ratio

$$C(f) = \|f * f\|_2^2 \;/\; \big(\|f * f\|_1\, \|f * f\|_\infty\big),$$

which serves as the fitness. The state-of-the-art solution found by *AlphaEvolve* (Georgiev et al., 2025) achieves $C(f) = 0.96102$ and is a highly irregular step function of 50 000 steps (Figure 5a), making this benchmark substantially more complex than hexagon packing.

The evolved *ImprovEvolve* program achieved $C(f) = 0.9512$, close to *AlphaEvolve*'s 0.96102 and surpassing all other open-source frameworks (Wang et al., 2025; Assumpção et al., 2026) (Table 1). The *GigaEvo* baseline reached only $C(f) = 0.9478$ under matched conditions (Figure 4 shows the single-run fitness distribution; Table 4 reports multi-run robustness). Figure 10 in Appendix B traces the full optimization path, showing how f and $f * f$ evolve across the two-stage validation scheme and how a single perturbation at $\varepsilon = 10^{-3}$ triggers a large jump in $C(f)$.

A key advantage of *ImprovEvolve*'s modular design is the built-in flexibility to resume optimization from a given high-quality solution. We demonstrate this with **ImprovEvolve+E**: starting from *AlphaEvolve*'s solution (Figure 5a), we apply a human-edited version of the evolved program with three changes (annotated diff in Section B.2.1): (i) replacing the original multi-grid schedule (resolutions up to 32 768) with a progressive schedule that begins at the input resolution and scales up to 1.6 million steps, so the optimizer can meaningfully refine *AlphaEvolve*'s 50 000-step function; (ii) removing the lower clipping at 10^{-10} that would otherwise erase small input-function steps; and (iii) raising the L-BFGS iteration cap to allow finer convergence at the higher resolutions. In just fifteen iterations of the `improve-perturb` loop we obtain a new state-of-the-art lower bound of $C(f) = 0.96258$ (Figure 5b). As with hexagon packing, this highlights the value of human–LLM collaboration: the LLM produces the core algorithmic structure through evolution, while a domain expert contributes targeted refinements that would be hard for the model to discover unaided—here, the insight that a much finer discretisation is needed to improve on a 50 000-step solution.

4.3. Robustness and Reproducibility

To confirm that the observed gains reflect an algorithmic advantage rather than favorable variance under the stochastic LLM evolution, we ran three independent 100-generation runs of both *GigaEvo* and *ImprovEvolve* under identical conditions. Table 4 reports the mean, sample standard deviation, and 95% confidence intervals, along with a Welch's t-test on the final fitness scores. *ImprovEvolve* achieves a statistically significant improvement on both HEX 11 ($p = 0.0086$) and ACI 2 ($p = 0.0493$), confirming that the modular decomposition delivers a reproducible advantage over learning a monolithic end-to-end optimizer.

4.4. Spherical codes

As a third benchmark, we apply *ImprovEvolve* to the *spherical codes* (or *Tammes*) problem (Cohn), which asks for an arrangement of N points on the unit sphere $S^{d-1} \subset \mathbb{R}^d$ that maximises the minimum pairwise distance—equivalently,

Table 4. Multi-run performance statistics across 3 independent runs per setting. Best single-run values are bolded. Welch's t-test confirms that *ImprovEvolve* yields a statistically significant mean improvement ($p < 0.05$) across runs.

Task	Method	Best	Mean ± Std	95% CI	Welch's t / p-value
HEX 11 (↓)	*GigaEvo* (baseline)	3.9296	3.9309 ± 0.0016	[3.9270, 3.9349]	$t = -3.96$
	ImprovEvolve	**3.9245**	**3.9261 ± 0.0014**	**[3.9227, 3.9295]**	**p = 0.0086**
ACI 2 (↑)	*GigaEvo* (baseline)	0.9478	0.9430 ± 0.0041	[0.9328, 0.9532]	$t = 2.43$
	ImprovEvolve	**0.9512**	**0.9494 ± 0.0019**	**[0.9447, 0.9541]**	**p = 0.0493**

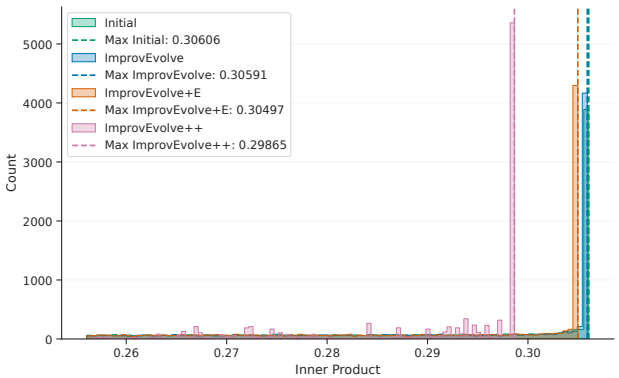


Figure 6. Right tail of the inner-product distribution for $(N, d) = (296, 16)$ when the `Improvers` are validated starting from the catalogue of Cohn. The maximum cosine $\mu(\mathcal{X})$ drops from 0.30606 (Cohn) to 0.30591 (*ImprovEvolve*), 0.30497 (*ImprovEvolve+E*), and 0.29865 (*ImprovEvolve++*, our calibrated general improver); the density at the new, tighter maximum simultaneously grows. Full distribution and UMAP of the final configurations: Section D.3 and Figure 11.

minimises the maximum cosine

$$\mu(\mathcal{X}) \;=\; \max_{i \neq j} \langle x_i, x_j \rangle, \qquad x_i \in S^{d-1},$$

with fitness $-\mu(\mathcal{X})$. Each configuration $\mathcal{X} = \{x_1, \ldots, x_N\}$ lives in an Nd-dimensional ambient space subject to N unit-norm constraints, yielding a highly non-convex landscape with rich symmetry structure. Exact optima are known only for $N \leq 2d$; for larger N, the online catalogue of Cohn maintains the best known configurations across a wide range of (N, d) and serves as an extensible benchmark accumulated over decades of expert effort. The closely related *kissing number* problem in $\mathbb{R}^d$ can be cast as deciding the feasibility of $\mu(\mathcal{X}) \leq 1/2$.

Setup. Evolution was run for the single configuration $(N, d) = (600, 11)$ with the *GigaEvo* defaults of Section 3 and the task prompt of Section D.5; the program with the highest fitness $-\mu(\mathcal{X})$ becomes the *ImprovEvolve* `Improver`, and asking *Gemini 3.1 Pro* to refine it under the minimal-edits prompt of Appendix C produces *ImprovEvolve+E* (diff and sources in Appendix D). A separate *GigaEvo* run, evolved jointly across a panel of configurations spanning $d \in [8, 16]$ and then tuned by the calibrated search protocol of Section D.2, yields the stronger *general* improver *ImprovEvolve++*. Validation replaces the $K = 10$ Stage-A seeds with the best-known catalogue configuration of Cohn and runs Stage-B basin-hopping with σ on a geometric schedule under monotone acceptance, directly exercising the resume-from-input interface; per-program protocols and budgets are deferred to Sections D.1 and D.2. The inner-product diagnostics of Figure 6 use the high-headroom configuration $(296, 16)$.

Table 5. Ablation study of the *ImprovEvolve* validation scheme on HEX 11. L: best side length; K: Stage A seeds; R: Stage B rounds. σ decreases geometrically from 100 to 0.001.

	A only	B only	A+B	A+B+LLM[1]	Lat.+B[2]
Best L	3.9282	3.9269	3.92**45**	3.9270	3.9310
Seeds K	1000	1	100	100	100
Rounds R	0	100	5	100	100

[1] *Gemini 3 Pro* was prompted to edit the evolved program (see Appendix C).
[2] Using `generate_config` produced by *Gemini 3 Pro*'s edited program.

Results. On the full benchmark of 90 catalogue codes spanning $d \in [8, 16]$ and $N \in [26, 1021]$, the calibrated general improver *ImprovEvolve++* strictly lowers the maximum cosine $\mu(\mathcal{X})$ below the best-known value on 53 of them, a mean relative improvement of $\mathbf{0.1765}\%$ when non-improvements are scored as zero (Table 6)—$2.35\times$ the best prior program (*ImprovEvolve* 0.0752%, *ImprovEvolve+E* 0.0618%), with the largest gains at $d = 13$ and $d = 16$. At the diagnostic configuration $(296, 16)$, the max cosine drops from 0.30606 (Cohn) to 0.29865 (*ImprovEvolve++*), and the density at the new, tighter maximum grows—more pairs sit at the smaller separation, signalling a structured, discrete-spectrum code (full distribution in Section D.3). A UMAP of the final configurations (Figure 11) shows that *ImprovEvolve* and *ImprovEvolve+E* barely move the points, whereas *ImprovEvolve++* relocates a substantial part of the configuration into structurally distinct regions, consistent with its larger gain. To put these figures in perspective, the maximum cosines of the best-known 11D configurations for $N = 593$ (the state-of-the-art 11D kissing number (Novikov et al., 2025)) and $N = 596$ differ by a mere 0.18%; even fractional-percentage improvements in the maximum cosine thus represent highly non-trivial structural rearrangements equivalent to accommodating multiple additional points.

5. Ablation study & Variants

5.1. Stage ablation

Both Stage A (multi-seed initialization) and Stage B (basin-hopping) are essential to the validation scheme. Table 5 reports an ablation on HEX 11. Stage A alone can pro-

duce a prototype of the optimal solution (see Figure 7 in Appendix A) but cannot refine it. Stage B alone can polish a single initialization, but without a diverse set of starting points, the probability of reaching the global optimum stays low.

Although basin-hopping with acceptance temperature $T > 0$ and a simulated-annealing schedule is theoretically preferable to the monotonic case ($T = 0$), its practical impact is limited: high temperatures prevent full refinement, while low temperatures barely improve the chance of escaping a local optimum. Worse, $T > 0$ risks abandoning a suboptimal configuration that lies on the path to the global optimum—for example, the $L = 4.6188$ arrangement for HEX 17 (Figure 7). Recovering such a configuration can be prohibitively slow, particularly for problems like ACI 2 where each `improve` call is expensive.

Can an LLM replace human tuning? The *ImprovEvolve+E* results relied on expert-guided hyperparameter edits, which raises the question of whether an LLM can match these autonomously. We prompted *Gemini 3 Pro* to make minimal, non-structural modifications to the HEX 11 programs—adjusting constants, parameter values, or function arguments only (full prompt in Appendix C). For the *GigaEvo* baseline (edits marked `LLM-EDIT` in Section A.2.1), the LLM *increased* Langevin noise to escape local minima and enlarged the batch size; the result was overlapping hexagons—the noise should have been *decreased*. For the *ImprovEvolve* program (Section A.2.1), edits included softening penalty weights and switching the lattice orientation from point- to flat-topped; while the latter helped dense packings at $n = 13, 21, 31$, it suppressed the stochastic exploration Stage A relies on, and both the fully edited program and a hybrid variant underperformed the unedited version (Table 5).

These results expose a common failure mode: even advanced LLMs suggest plausible-sounding but empirically harmful edits. The damage is exacerbated in monolithic programs and mitigated by modular ones. Evolution can partially compensate by feeding validation results back into context, but models still lack a reliable sense for choosing optimization hyperparameters; agentic loops that evaluate candidate edits under different configurations are a more promising route.

6. Discussion and Conclusion

ImprovEvolve demonstrates that decomposing an optimization task into modular subproblems substantially improves the output of an evolutionary coding agent. Decomposition is a well-established principle, but its significance for LLM-powered discovery deserves emphasis. Current models produce a complete response in a single turn regardless of problem difficulty, while task decomposition, already a staple of AI-assisted code editors, can bring comparable benefit to AI for mathematics, with researchers specifying the modules the program must contain and agents validating and debugging them so the LLM-guided evolution can focus on the conceptual core.

Decomposition is especially important in mathematics, where a promising idea may give a consistently poor signal until its implementation is fully correct. For example, reproducing the 592-sphere 11D kissing number construction took roughly 20 chat exchanges with *Gemini 3 Pro* provided with the LaTeXsource of the original article (Ganzhinov, 2022), and intermediate versions yielded fewer than 100 valid spheres. Such a complex construction is unlikely to arise through evolution alone, because the sparse reward signal would discard the idea before a correct implementation is obtained. Human experts can tailor the decomposition, for example, by delegating non-local search to basin-hopping. Combining these tailored decompositions with agentic workflows that autonomously decide which modules to refine and when to invoke external solvers represents a promising direction for scaling LLM-powered mathematical discovery.

Limitations

Our main quantitative evaluation is limited to three mathematical optimization problems: hexagon packing, the second autocorrelation inequality, and spherical codes. While the results on the reported tasks are strong, broader generalisability to other problem classes remains to be demonstrated. Beyond the multi-run study in Table 4, the headline state-of-the-art numbers are single-run best results and may vary across runs given the stochastic nature of evolution and basin-hopping.

Impact Statement

This paper aims to advance machine learning by applying LLM-guided evolutionary methods to mathematical optimization. The primary ethical consideration is computational cost: each evolutionary run consumes significant energy and API resources. We mitigate this with a two-tier LLM choice strategy and a modular framework that reduces the iterations needed to reach competitive results. Broader societal impacts beyond standard scientific research are not expected to differ materially from other optimization and automated-discovery work.

References

Assumpção, H., Ferreira, D., Campos, L., and Murai, F. CodeEvolve: an open source evolutionary coding agent for algorithm discovery and optimization, 2026. URL

https://arxiv.org/abs/2510.14150.

Berthold, T., Kamp, D., Mexi, G., Pokutta, S., and Pólik, I. Global optimization for combinatorial geometry problems revisited in the era of LLMs, 2026. URL https://arxiv.org/abs/2601.05943. Additional solver results reported by the authors at: https://web.archive.org/web/20260216061025/https://erich-friedman.github.io/packing/hexinhex/.

Cohn, H. Spherical codes. https://cohn.mit.edu/spherical-codes/. Online catalogue of best known spherical codes; accessed 2026-05.

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Friedman, E. Hexagons in hexagons — Erich's packing center. https://web.archive.org/web/20250716213831/https://erich-friedman.github.io/packing/hexinhex/, 2025. Archived: 2025-07-16.

Ganzhinov, M. Highly symmetric lines, 2022. URL https://arxiv.org/abs/2207.08266.

Georgiev, B., Gómez-Serrano, J., Tao, T., and Wagner, A. Z. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Gideoni, Y., Tang, Y., Risi, S., and Gal, Y. Random baselines for simple code problems are competitive with code evolution. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. URL https://openreview.net/forum?id=rbVIpbmbTc.

Jaech, A. and Joseph, A. Further improvements to the lower bound for an autoconvolution inequality, 2025. URL https://arxiv.org/abs/2508.02803.

Khrulkov, V., Galichin, A., Bashkirov, D., Vinichenko, D., Travkin, O., Alferov, R., Kuznetsov, A., and Oseledets, I. GigaEvo: An open source optimization framework powered by llms and evolution algorithms, 2025. URL https://arxiv.org/abs/2511.17592.

Lange, R. T., Imajuku, Y., and Cetin, E. ShinkaEvolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349.

Mouret, J.-B. and Clune, J. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J., Mehrabian, A., et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Sharma, A. OpenEvolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Wales, D. J. and Doye, J. P. K. Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, July 1997. ISSN 1520-5215. doi: 10.1021/jp970984n. URL http://dx.doi.org/10.1021/jp970984n.

Wang, Y., Su, S.-R., Zeng, Z., Xu, E., Ren, L., Yang, X., Huang, Z., He, X., Ma, L., Peng, B., Cheng, H., He, P., Chen, W., Wang, S., Du, S. S., and Shen, Y. ThetaEvolve: Test-time learning on open problems, 2025. URL https://arxiv.org/abs/2511.23473.

Yan, M., Peng, B., Coleman, B., Chen, Z., Xie, Z., He, Z., Sachdeva, N., Ye, I., Wang, W., Wang, C., Chi, E. H., Kang, W.-C., Cheng, D. Z., and Wang, B. PACEvolve: Enabling long-horizon progress-aware consistent evolution, 2026. URL https://arxiv.org/abs/2601.10657.

Yuksekgonul, M., Koceja, D., Li, X., Bianchi, F., McCaleb, J., Wang, X., Kautz, J., Choi, Y., Zou, J., Guestrin, C., et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

A. Hexagon Packing Problem: Task Description and Evolved Solution

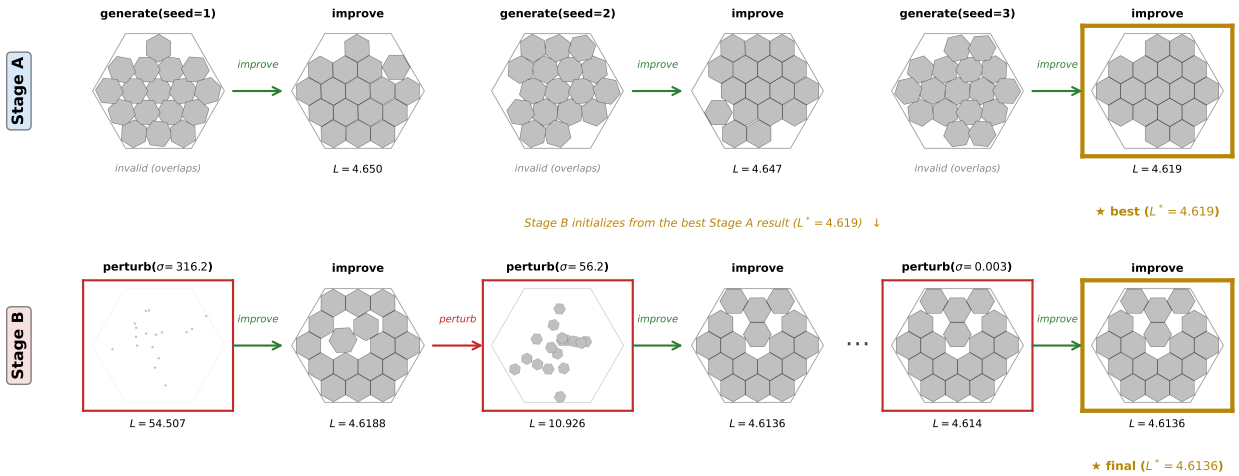


Figure 7. Illustration of the two-stage validation scheme (Algorithm 1) on the HEX 17 problem. *Stage A (top):* `generate` proposes initial configurations that typically contain overlapping hexagons (invalid); `improve` resolves these overlaps and produces valid packings. The best result ($L^* = 4.619$, gold border) is selected. *Stage B (bottom):* basin-hopping alternates `perturb` (red border) and `improve`. Even an extreme perturbation ($\sigma = 316$, round 4) that scatters hexagons across a vast enclosing hexagon ($L = 54.5$) is recovered by `improve` back to $L = 4.619$. The breakthrough occurs at round 8 ($\sigma = 56.2$), where `improve` discovers a new, tighter basin at $L = 4.6136$—a structural improvement inaccessible from the Stage A initialization. Subsequent rounds with small σ confirm stability.

A.1. Prompts

A.1.1. Baseline

TASK DEFINITION – HEXAGON PACKING PROBLEM

Challenge: Implement a Python function that minimizes the side length of a flat-topped regular enclosing hexagon required to contain N non-overlapping unit regular hexagons.

Specific Benchmark: $N = 11$.
General Requirement: The code should be valid for any N from 1 to 25, defaulting to 11.

GEOMETRIC SPECIFICATIONS

1. **Unit Hexagons (Items):**
   - Regular hexagons with circumradius $r = 1.0$.
   - Side length $= 1.0$.
   - May be arbitrarily placed (x, y) and rotated (θ).
2. **Enclosing Hexagon (Container):**
   - Regular, flat-topped, centered at $(0, 0)$.
   - "Flat-topped" vertices are located at angles $[0°, 60°, 120°, 180°, 240°, 300°]$ relative to the center.
   - Objective: Minimize the enclosing hexagon's side length L (which is equal to its circumradius).

A.1.1. With Improver

TASK DEFINITION – HEXAGON PACKING OPTIMIZER

Challenge: Implement a Python class that minimizes the side length of a flat-topped regular enclosing hexagon required to contain N non-overlapping unit regular hexagons.

Specific Benchmark: $N = 11$.
General Requirement: The code must be valid for any N from 1 to 25.

GEOMETRIC SPECIFICATIONS

1. **Unit Hexagons (Items):**
   - Regular hexagons with circumradius $r = 1.0$.
   - Side length $= 1.0$.
   - May be arbitrarily placed (x, y) and rotated (θ).
2. **Enclosing Hexagon (Container):**
   - Regular, flat-topped, centered at $(0, 0)$.
   - "Flat-topped" vertices are located at angles $[0°, 60°, 120°, 180°, 240°, 300°]$ relative to the center.
   - Objective: Minimize the enclosing hexagon's side length L (which is equal to its circumradius).

OBJECTIVE FUNCTION
Maximize Fitness: $-L$ (Minimize side length L).
Target for $N = 11$: $L \leq 3.92$.

CONSTRAINTS

1. **Containment:** All vertices of every unit hexagon must lie inside or on the boundary of the enclosing hexagon.
2. **Non-overlap:** The intersection area between any pair of unit hexagons must be zero (touching is allowed).
3. **Count:** Exactly `hex_num` unit hexagons must be placed.

IMPLEMENTATION REQUIREMENTS

Libraries:
Use `numpy`, `scipy`, `sklearn`, `jax`, `optax`, `jaxopt` or standard Python libraries.

Function Interface:
You must implement the following function structure exactly:

```
import numpy as np

def entrypoint(hex_num=11, seed=0) -> tuple[np.ndarray,
 np.ndarray]:
    """
    Calculates the optimal packing configuration.

    Args:
        hex_num: Number of hexagons to pack.
        seed: Random seed for reproducibility.

    Returns:
        centers: np.ndarray of shape (hex_num, 2)
representing (x, y) coordinates.
        angles: np.ndarray of shape (hex_num,) in
radians [0, 2pi).
    """
    # Implement optimization logic here
    pass
```

CRITICAL FAILURE MODES

- **Wrong Shapes:** Returning arrays of shape `(12, ...)` when `hex_num=11`.
- **Geometry Errors:** Assuming point-topped enclosing hexagon instead of flat-topped.
- **Invalid Output:** Hexagons overlapping or protruding outside the boundary in the final result.
- **Wrong Return Type:** Returning a class or object instead of the `(centers, angles)` tuple.

DOCUMENTATION
Supply your code with a brief explanation of the algorithm using **LaTeX** in the introductory comment.

OBJECTIVE FUNCTION
Maximize Fitness: $-L$ (Minimize side length L).
Target for $N = 11$: $L \leq 3.92$.

CONSTRAINTS

1. **Containment:** All vertices of every unit hexagon must lie inside or on the boundary of the enclosing hexagon.
2. **Non-overlap:** The intersection area between any pair of unit hexagons must be zero (touching is allowed).
3. **Count:** Exactly `hex_num` unit hexagons must be placed.

IMPLEMENTATION REQUIREMENTS

Libraries:
Use `numpy`, `scipy`, `sklearn`, `jax`, `optax`, `jaxopt` or standard Python libraries.

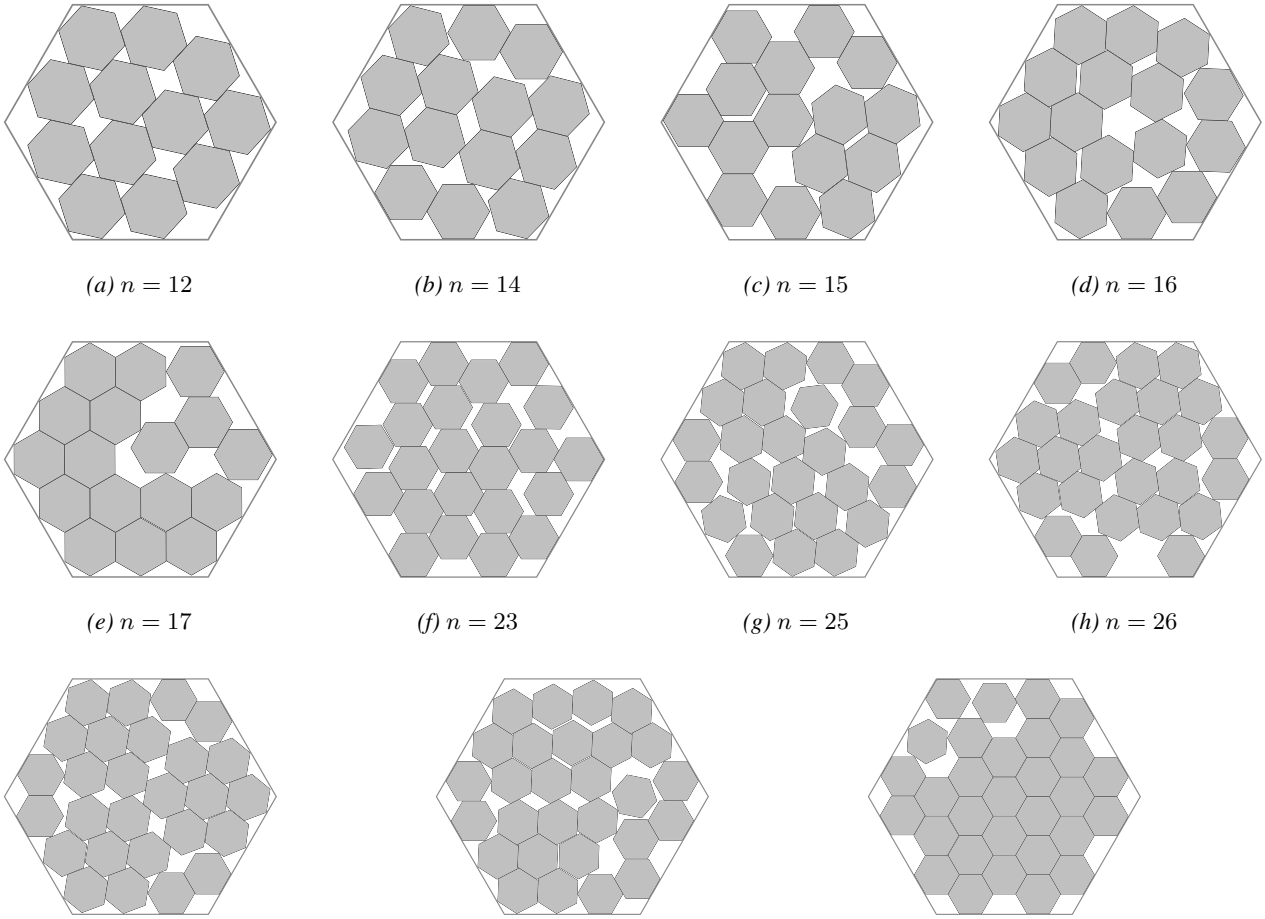
Class Interface:
You must implement the following class structure exactly:

```
import numpy as np

class Improver:
    def __init__(self, hex_num=11, seed: int = 0):
        self.hex_num = hex_num
        self.seed = seed

    def improve(self, input_config: tuple[np.ndarray,
np.ndarray], seed=None) -> tuple[np.ndarray, np.ndarray
]:
        """
        Refines the configuration to minimize L.
        """
        pass

    def perturb(self, input_config, intensity: float,
seed=None) -> tuple[np.ndarray, np.ndarray]:
        """
        Perturbs the configuration (position/rotation).
        """
        pass

    def generate_config(self, seed=None) -> tuple[np.
ndarray, np.ndarray]:
        """
        Generates a valid starting configuration.
        """
        pass

def entrypoint():
    return Improver
```

CRITICAL FAILURE MODES

- **Wrong Shapes:** Returning arrays of shape `(12, ...)` when `hex_num=11`.
- **Geometry Errors:** Assuming point-topped enclosing hexagon instead of flat-topped.
- **Invalid Output:** Hexagons overlapping or protruding outside the boundary in the final result.

DOCUMENTATION
Supply your code with a brief explanation of the algorithm using **LaTeX** in the introductory comment.



(a) $n = 12$ *(b)* $n = 14$ *(c)* $n = 15$ *(d)* $n = 16$

(e) $n = 17$ *(f)* $n = 23$ *(g)* $n = 25$ *(h)* $n = 26$

(i) $n = 27$ *(j)* $n = 28$ *(k)* $n = 29 - 30$

Figure 8. Extended state-of-the-art hexagon packings discovered by *ImprovEvolve* and *ImprovEvolve+E*. New best-known configurations are shown for $n = 12$–17 and $n = 23$. For $n = 25$–30, no prior packings have been reported; the discovered configurations exhibit structured, non-chaotic arrangements. The $n = 30$ packing (shown) achieves the same side length $L = 6.0$ as $n = 29$: any single hexagon can be removed to obtain a valid $n = 29$ packing with identical L. Side lengths are listed in Table 1 and Table 3.

A.2. Evolved Solution Code

A.2.1. Baseline

The following program was evolved by *GigaEvo* (Program ID: 80ed10a3-7d87-42f7-a610-068c6d69411e, Fitness: -3.9327, Generation: 13).

```
import numpy as np
import jax
import jax.numpy as jnp
import optax

def entrypoint(hex_num=11, seed=0) -> tuple[np.ndarray,
↪  np.ndarray]:
    """"
    Algorithm: Gradient-based packing optimization with
    ↪  SAT for non-overlap
    and explicit projection constraints for a
    ↪  flat-topped enclosing hexagon.
    L is the side length (and circumradius) of the
    ↪  container.
    """"
```

A.2.1. With Improver

The following program was evolved by *ImprovEvolve* (Program ID: e98264be-75e4-4160-aaaa-530ec55f7848, Fitness: -3.92467, Generation: 13). Lines marked with `EDIT` show the human modifications applied in the *ImprovEvolve+E* variant: the optimizer was changed from `L-BFGS-B` to `SLSQP`, initial penalty weights were softened, variable bounds were relaxed, and the iteration limit was increased.

```
import numpy as np
import jax
import jax.numpy as jnp
from scipy.optimize import minimize

jax.config.update("jax_enable_x64", True)

@jax.jit
def get_unit_hex_verts(center, theta):
```

```python
jax.config.update("jax_enable_x64", True)

# Flat-topped container normals at 30, 90, 150, 210, 270, 330 degrees
cont_angles = jnp.deg2rad(jnp.array([30, 90, 150, 210, 270, 330]))
container_normals = jnp.stack([jnp.cos(cont_angles), jnp.sin(cont_angles)], axis=1)

# Unit hexagon local geometry
unit_angles = jnp.arange(6) * (jnp.pi / 3.0)
unit_verts_local = jnp.stack([jnp.cos(unit_angles), jnp.sin(unit_angles)], axis=1)
sat_axes_local = jnp.array([0, jnp.pi/3, 2*jnp.pi/3]) + jnp.pi/6  # Midpoints of edges

def get_world_data(centers, angles):
    c, s = jnp.cos(angles), jnp.sin(angles)
    rot = jnp.stack([jnp.stack([c, -s], axis=-1), jnp.stack([s, c], axis=-1)], axis=-2)
    verts = jnp.einsum('nij,kj->nki', rot, unit_verts_local) + centers[:, None, :]
    axes_angles = angles[:, None] + sat_axes_local[None, :]
    axes = jnp.stack([jnp.cos(axes_angles), jnp.sin(axes_angles)], axis=2)
    return verts, axes

def containment_loss(verts, L):
    # For flat-topped hexagon with side L, apothem is L * cos(30 deg)
    h = L * jnp.cos(jnp.pi / 6.0)
    # Projections of unit hexagon vertices onto container normals must be <= h
    projections = jnp.dot(verts.reshape(-1, 2), container_normals.T)
    return jnp.sum(jax.nn.relu(projections - h)**2)

def overlap_loss(verts, axes_all):
    N = verts.shape[0]
    idx_i, idx_j = jnp.triu_indices(N, 1)

    def pair_penalty(i, j):
        v_i, v_j = verts[i], verts[j]
        axes = jnp.concatenate([axes_all[i], axes_all[j]], axis=0)
        p_i = jnp.dot(v_i, axes.T)
        p_j = jnp.dot(v_j, axes.T)
        gaps = jnp.maximum(jnp.min(p_j, axis=0) - jnp.max(p_i, axis=0),
                           jnp.min(p_i, axis=0) - jnp.max(p_j, axis=0))
        return jax.nn.relu(-jnp.max(gaps) + 1e-9)**2

    return jnp.sum(jax.vmap(pair_penalty)(idx_i, idx_j))

def optimize_instance(sub_key):
    total_steps = 22000
    # LLM-EDIT: total_steps = 30000
    k1, k2, k3 = jax.random.split(sub_key, 3)

    # Insight: Expanded range based on expected L ~ 4.0
    centers = jax.random.uniform(k1, (hex_num, 2), minval=-3.5, maxval=3.5)
    # LLM_EDIT: centers = jax.random.uniform(k1, (hex_num, 2), minval=-3.0, maxval=3.0)
    init_angles = jax.random.uniform(k2, (hex_num,), minval=0.0, maxval=jnp.pi/3.0)
    params = {'centers': centers, 'angles': init_angles, 'log_L': jnp.log(4.5)}

    lr = optax.cosine_decay_schedule(0.04, total_steps, alpha=1e-3)
    optimizer = optax.chain(optax.clip_by_global_norm(1.0), optax.adam(learning_rate=lr))
    opt_state = optimizer.init(params)
```

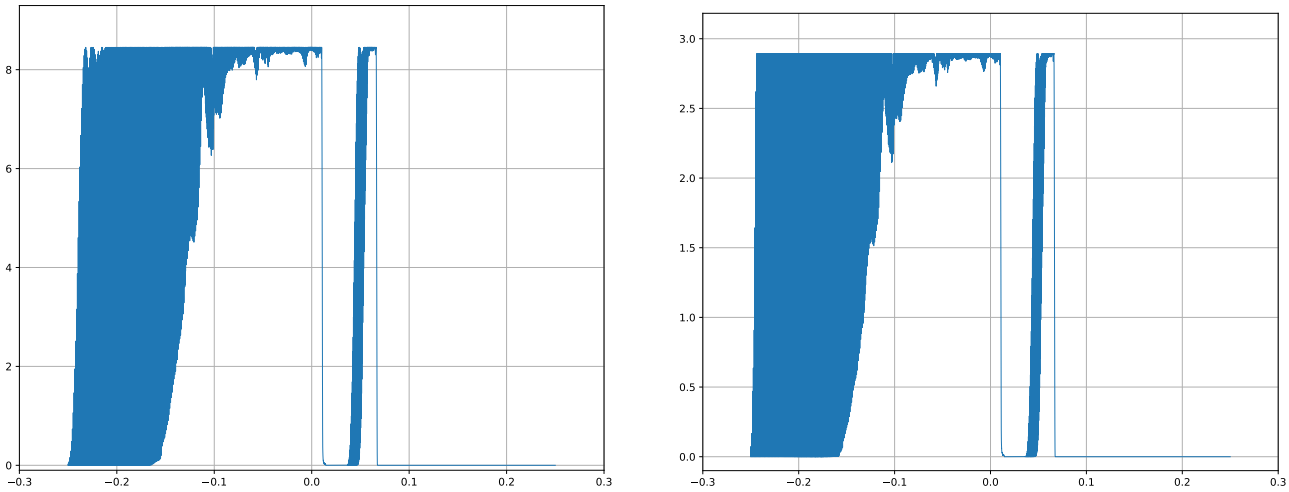
```python
    # Regular hexagon vertices, circumradius 1.0
    base_angles = jnp.array([0.0, jnp.pi/3, 2*jnp.pi/3, jnp.pi, 4*jnp.pi/3, 5*jnp.pi/3], dtype=jnp.float64)
    angles = base_angles + theta
    return center + jnp.stack([jnp.cos(angles), jnp.sin(angles)], axis=1)

@jax.jit
def get_container_normals():
    # Normals for flat-topped enclosing hexagon (face-centered directions)
    angles = jnp.array([jnp.pi/6, jnp.pi/2, 5*jnp.pi/6, 7*jnp.pi/6, 3*jnp.pi/2, 11*jnp.pi/6], dtype=jnp.float64)
    return jnp.stack([jnp.cos(angles), jnp.sin(angles)], axis=1)

@jax.jit
def compute_separation(c1, theta1, c2, theta2):
    v1 = get_unit_hex_verts(c1, theta1)
    v2 = get_unit_hex_verts(c2, theta2)
    axes_angles = jnp.array([jnp.pi/6, jnp.pi/2, 5*jnp.pi/6], dtype=jnp.float64)
    n1 = jnp.stack([jnp.cos(axes_angles + theta1), jnp.sin(axes_angles + theta1)], axis=1)
    n2 = jnp.stack([jnp.cos(axes_angles + theta2), jnp.sin(axes_angles + theta2)], axis=1)
    axes = jnp.concatenate([n1, n2], axis=0)
    p1 = jnp.dot(v1, axes.T)
    p2 = jnp.dot(v2, axes.T)
    gaps = jnp.maximum(jnp.min(p1, axis=0) - jnp.max(p2, axis=0), jnp.min(p2, axis=0) - jnp.max(p1, axis=0))
    return jnp.max(gaps)

@jax.jit
def objective_fn(params, idx_i, idx_j, weight_cont, weight_ov, margin):
    N = (len(params) - 1) // 3
    centers = params[:2*N].reshape((N, 2))
    thetas = params[2*N:3*N]
    L = params[-1]

    # Flat-topped container containment
    H = L * (jnp.sqrt(3.0) / 2.0)
    normals = get_container_normals()

    def hex_cont(c, t):
        v = get_unit_hex_verts(c, t)
        return jnp.sum(jax.nn.relu(jnp.dot(v, normals.T) - H)**2)

    total_cont = jnp.sum(jax.vmap(hex_cont)(centers, thetas))

    def hex_ov(i, j):
        sep = compute_separation(centers[i], thetas[i], centers[j], thetas[j])
        return jax.nn.relu(margin - sep)**2

    total_ov = jnp.sum(jax.vmap(hex_ov)(idx_i, idx_j))

    return L + weight_cont * total_cont + weight_ov * total_ov

_grad_jit = jax.jit(jax.value_and_grad(objective_fn))

class Improver:
    def __init__(self, hex_num=11, seed: int = 0):
        self.hex_num = hex_num
        self.seed = seed
        self.idx_i, self.idx_j = map(jnp.array, np.triu_indices(hex_num, k=1))

    def generate_config(self, seed=None) -> tuple[np.ndarray, np.ndarray]:
        rng = np.random.default_rng(seed if seed is not None else self.seed)
        centers = []
        for q in range(-5, 6):
```

```
        def step_fn(carry, i):
            p, state, rng = carry
            prog = i / total_steps
            # Smoother, more gradual weight ramping to
            ↪  allow rearrangement
            weight = 50.0 + 5e6 * jnp.power(prog, 2.8)

            def loss_fn(par):
                L = jnp.exp(par['log_L'])
                # Maintain symmetry constraint
                ang = par['angles'] % (jnp.pi / 3.0)
                v, a = get_world_data(par['centers'],
                ↪  ang)
                c_err = containment_loss(v, L)
                o_err = overlap_loss(v, a)
                return L + weight * (c_err + 5.0 *
                ↪  o_err), (L, c_err + o_err)

            (val, (L, err)), grads =
            ↪  jax.value_and_grad(loss_fn,
            ↪  has_aux=True)(p)

            rng, n_rng = jax.random.split(rng)
            # Langevin-style noise for escaping local
            ↪  minima
            temp = 1e-3 * jnp.exp(-4.0 * prog)
            # LLM-EDIT: temp = 2e-3 * jnp.exp(-3.5 *
            ↪  prog)
            grads['centers'] += jax.random.normal(n_rng,
            ↪  grads['centers'].shape) * temp

            updates, state = optimizer.update(grads,
            ↪  state)
            p = optax.apply_updates(p, updates)
            return (p, state, rng), (L, err)

        (final_p, _, _), (h_L, h_e) =
        ↪  jax.lax.scan(step_fn, (params, opt_state,
        ↪  k3), jnp.arange(total_steps))
        return final_p, h_L[-1], h_e[-1]

    batch_size = 96
    # LLM-EDIT: batch_size = 256
    rng_key = jax.random.PRNGKey(seed)
    keys = jax.random.split(rng_key, batch_size)
    all_p, all_L, all_e =
    ↪  jax.jit(jax.vmap(optimize_instance))(keys)

    # Precision check
    valid = all_e < 1e-7
    best_idx = jnp.where(valid, all_L, 1e10).argmin()
    best_idx = jax.lax.cond(valid[best_idx], lambda:
    ↪  best_idx, lambda: all_e.argmin())

    res = jax.tree_util.tree_map(lambda x: x[best_idx],
    ↪  all_p)
    return np.array(res['centers']),
    ↪  np.array(res['angles']) % (2*np.pi)
```

```
            for r in range(-5, 6):
                if abs(q+r) <= 5:
                    # Hexagonal lattice base
                    x = (q + r/2.0) * np.sqrt(3)
                    y = r * 1.5
                    # LLM-EDIT: x = q * 1.5
                    # LLM-EDIT: y = (r + q/2.0) *
                    ↪  np.sqrt(3)
                    centers.append([x, y])
        centers = np.array(centers)
        dists = np.linalg.norm(centers, axis=1)
        # Start with slightly jittered central points
        idx = np.argsort(dists + rng.uniform(0, 0.5,
        ↪  size=len(dists)))[:self.hex_num]
        return centers[idx], rng.uniform(0, np.pi/3,
        ↪  self.hex_num)
        # LLM-EDIT: return centers[idx], rng.normal(0,
        ↪  0.01, self.hex_num)

    def perturb(self, input_config, intensity,
    ↪  seed=None):
        rng = np.random.default_rng(seed if seed is not
        ↪  None else self.seed)
        c, a = map(np.copy, input_config)
        if intensity > 2.0:
            # Structural swap to escape local optima
            i, j = rng.choice(self.hex_num, 2,
            ↪  replace=False)
            c[i], c[j] = c[j].copy(), c[i].copy()
            a[i], a[j] = a[j].copy(), a[i].copy()

        c += rng.normal(0, 0.05 * intensity, c.shape)
        a += rng.normal(0, 0.02 * intensity, a.shape)
        return c, np.mod(a, np.pi/3)

    def improve(self, input_config, seed=None):
        centers, angles = input_config
        L_est = np.max(np.linalg.norm(centers, axis=1))
        ↪  + 1.0
        x = np.concatenate([centers.flatten(), angles,
        ↪  [L_est]]).astype(np.float64)

        # Graduated hardening schedule for N=11
        ↪  precision
        stages = [
            # EDIT: {'w_cont': 1e2, 'w_ov': 10,
            ↪  'margin': 1e-3}
            {'w_cont': 1e3, 'w_ov': 1e2, 'margin':
            ↪  1e-3},
            {'w_cont': 1e4, 'w_ov': 1e4, 'margin':
            ↪  1e-4},
            {'w_cont': 1e6, 'w_ov': 1e6, 'margin':
            ↪  1e-5},
            {'w_cont': 1e9, 'w_ov': 1e8, 'margin':
            ↪  1e-7}
        ]
        # LLM-EDIT:
        # stages = [
        #     {'w_cont': 1e1, 'w_ov': 1e1, 'margin':
        ↪  1e-1},
        #     {'w_cont': 1e3, 'w_ov': 1e3, 'margin':
        ↪  1e-3},
        #     {'w_cont': 1e6, 'w_ov': 1e6, 'margin':
        ↪  1e-5},
        #     {'w_cont': 1e9, 'w_ov': 1e9, 'margin':
        ↪  1e-7}
        # ]

        for st in stages:
            res = minimize(
                lambda p: tuple(map(np.array,
                ↪  _grad_jit(p, self.idx_i, self.idx_j,
                ↪  st['w_cont'], st['w_ov'],
                ↪  st['margin']))),
                x, method='L-BFGS-B', jac=True,
                # EDIT: x, method='SLSQP', jac=True,
                bounds=[(-15, 15)]*(2*self.hex_num) +
                ↪  [(0.0, np.pi/3)]*self.hex_num +
                ↪  [(1.0, 15.0)],
```

```
                # EDIT: bounds=[(None,
                ↪  None)]*(2*self.hex_num) + [(None,
                ↪  None)]*self.hex_num + [(1.0,
                ↪  15.0)],
                options={'maxiter': 800, 'ftol': 1e-10}
                # EDIT: options={'maxiter': 10000,
                ↪  'ftol': 1e-11}
                # LLM-EDIT: options={'maxiter': 1000,
                ↪  'ftol': 1e-10}

            )
            x = res.x

        final_centers =
        ↪  x[:2*self.hex_num].reshape((self.hex_num,
        ↪  2))
        final_angles =
        ↪  np.mod(x[2*self.hex_num:3*self.hex_num],
        ↪  2*np.pi)
        return final_centers, final_angles

def entrypoint():
    return Improver
```

B. Second Autocorrelation Inequality: Prompt and Evolved Solution



(a) AlphaEvolve *(b) ImprovEvolve+E*

Figure 9. Autoconvolutions $f * f$ of the extremal functions from Figure 5. The *ImprovEvolve+E* solution achieves a higher ratio $C(f) = \|f * f\|_2^2/(\|f * f\|_1 \|f * f\|_\infty)$.

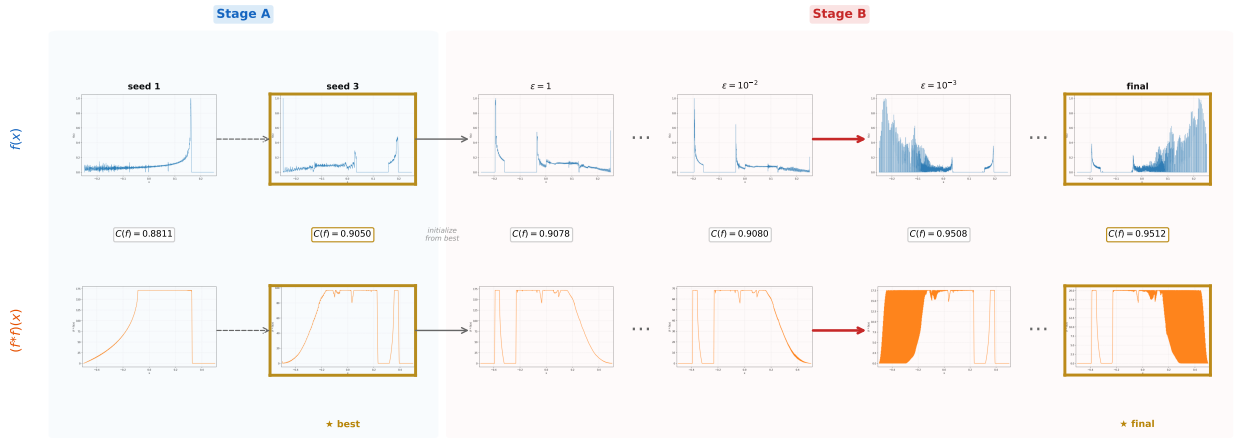


Figure 10. Optimization path of the *ImprovEvolve* validation scheme on the ACI 2 problem. Each column shows the evolved function $f(x)$ (top row) and its autoconvolution $(f*f)(x)$ (bottom row) at a key moment; $C(f)$ values are displayed between the rows. *Stage A* (blue background): two random seeds are generated and improved; seed 3 yields the best initial result ($C(f) = 0.9050$, gold border). *Stage B* (red background): the `perturb-improve` cycle is applied with geometrically decreasing perturbation intensity ε. Iterations at $\varepsilon = 1$ and $\varepsilon = 10^{-2}$ produce only marginal gains ($C(f) \approx 0.908$). At $\varepsilon = 10^{-3}$ (red arrow), $C(f)$ jumps to 0.9508, accompanied by a qualitative change in both f and $f*f$. Further refinement yields the final result of $C(f) = 0.9512$ (gold border).

B.1. Prompt

B.1.1. Baseline

TASK DEFINITION – SECOND AUTOCORRELATION INEQUALITY

Challenge: Functional optimization in analysis. Implement a Python function that generates a non-negative function $f : \mathbb{R} \to [0, \infty)$ to maximize the constant C in the second autocorrelation inequality.

Adaptive Resolution: The solver is **allowed and encouraged** to choose the optimal resolution (array size N) of the function.

- *Coarse Grids (Lower N):* faster iteration and global structure search.
- *Fine Grids (Higher N):* higher precision and higher potential fitness C.
- Your solution should ideally be scalable for N up to 100,000.

MATHEMATICAL SPECIFICATIONS

1. **The Function (f):**
   - A discrete 1D array representing a function $f(x)$ on a finite support.
   - **Non-negativity:** $f(x) \geq 0$ for all x.
   - **Non-triviality:** $\sum f(x) > 0$.
2. **The Convolution (g):**
   - $g = f \star f$. In the discrete domain, this is the discrete convolution of the array f with itself (linear convolution).
3. **The Objective:**
   - Maximize the ratio involving the L_1, L_2, and L_∞ norms of the autocorrelation g.

OBJECTIVE FUNCTION
Maximize the constant C:

$$C(f) = \frac{\|f \star f\|_2^2}{\|f \star f\|_1 \|f \star f\|_\infty}$$

Target: $C \geq 0.962$.

CONSTRAINTS

1. **Validity:** All elements of the output array must be non-negative finite floats.

B.1.1. For Improver

TASK DEFINITION – SECOND AUTOCORRELATION INEQUALITY

Challenge: Functional optimization in analysis. Implement a Python class that generates a non-negative function $f : \mathbb{R} \to [0, \infty)$ to maximize the constant C in the second autocorrelation inequality.

Adaptive Resolution: The solver is **allowed and encouraged** to change the resolution (array size N) of the function.

- *Coarse Grids (Lower N):* faster iteration and global structure search.
- *Fine Grids (Higher N):* higher precision and higher potential fitness C.
- Your solution must be scalable for N up to $100\,000$ (but, of course, with larger time limits).

MATHEMATICAL SPECIFICATIONS

1. **The Function (f):**
   - A discrete 1D array representing a function $f(x)$ on a finite support.
   - **Non-negativity:** $f(x) \geq 0$ for all x.
   - **Non-triviality:** $\sum f(x) > 0$.
2. **The Convolution (g):**
   - $g = f \star f$. In the discrete domain, this is the discrete convolution of the array f with itself (linear convolution).
3. **The Objective:**
   - Maximize the ratio involving the L_1, L_2, and L_∞ norms of the autocorrelation g.

OBJECTIVE FUNCTION
Maximize the constant C:

$$C(f) = \frac{\|f \star f\|_2^2}{\|f \star f\|_1 \|f \star f\|_\infty}$$

Target: $C \geq 0.962$.

CONSTRAINTS

1. **Validity:** All elements of the output array must be non-negative finite floats.

2. **Bounds:** $0.961 \leq C \leq 1$.
3. **Minimum Resolution:** The final output array must have length $N \geq 1024$ to ensure the discretization approximates the continuous function reasonably well.

IMPLEMENTATION REQUIREMENTS

Libraries:
Use `numpy`, `jax`, `optax`, `scipy`, `jaxopt`, or standard Python libraries. **JAX is preferred** for automatic differentiation and GPU acceleration.

Interface:
You must implement the following function structure exactly:

```
import numpy as np

def entrypoint() -> np.ndarray:
    """
    Generates and optimizes a function f to maximize
the autocorrelation constant C.

    The function should perform the optimization
internally (using JAX/SciPy/etc.)
    and return the final optimized 1D array.

    Returns:
        f: 1D NumPy array (float) representing the
optimized function.
           Shape should be at least (1024,).
    """
    pass
```

CRITICAL FAILURE MODES

- **Zero Function:** Returning an array of all zeros.
- **Negative Values:** Returning $f(x) < 0$.
- **Dimensionality Errors:** Returning a 2D array.
- **Excessively Low Resolution:** Returning $N < 1000$ where convolution properties degrade significantly.
- **Delusions:** Trying to "guess" optimal function rather than searching for it. Believe me, the example for $C = 0.961$ is a really strange function.

DOCUMENTATION

Supply your code with a brief explanation of the algorithm, using **LaTeX** in the introductory comment.

2. **Bounds:** $0.961 \leq C \leq 1$.
3. **Minimum Resolution:** The final output array must have length $N \geq 1024$ to ensure the discretization approximates the continuous function reasonably well.

IMPLEMENTATION REQUIREMENTS

Libraries:
Use `numpy`, `jax`, `optax`, `scipy`, `jaxopt`, or standard Python libraries. **JAX is preferred** for automatic differentiation and GPU acceleration.

Class Interface:
You must implement the following class structure exactly:

```
import numpy as np

class Improver:
    def __init__(self, seed: int = 0):
        """
        Initialize with reproducible random state.
        """
        self.seed = seed
        # Initialize JAX keys or RNG here

    def improve(self, input_f: np.ndarray) -> np.
ndarray:
        """
        Refines an existing function f using rigorous
local optimization.

        ADAPTIVE RESOLUTION:
        The output array does NOT need to match the
input size.

        Args:
            input_f: 1D NumPy array of shape (N_in,)

        Returns:
            improved_f: 1D NumPy array of shape (N_out,)

        """
        pass

    def perturb(self, input_f: np.ndarray, intensity:
float) -> np.ndarray:
        """
        Applies DISCRETE, structural, or RESOLUTION
modifications.
        Args:
            input_f: 1D NumPy array.
            intensity: Float (1e-3 to 1e3).

        Returns:
            perturbed_f: 1D NumPy array (size may
differ from input).
        """
        pass

    def generate_config(self, initial_resolution: int =
 1000) -> np.ndarray:
        """
        Generates a valid starting function f with a
maximum possible fitness.

        Args:
            initial_resolution: Suggested starting size
 N.

        Returns:
            f: 1D NumPy array of shape (
initial_resolution,)
        """
        pass
```

```
def entrypoint():
    return Improver
```

CRITICAL FAILURE MODES

- **Zero Function:** Returning an array of all zeros.
- **Negative Values:** Returning $f(x) < 0$.
- **Dimensionality Errors:** Returning a 2D array.
- **Excessively Low Resolution:** Returning $N < 1000$ where convolution properties degrade significantly.
- **Excessively High Resolution:** Returning $N > 20000$ where optimization is very slow. Your solution must work even for this case, but we will not validate it now, so put in the comments to your program how it should be scaled or program a "self.scale_mode" for your class.
- **Inefficiency:** Ignoring the benefits of Multigrid strategies (not utilizing lower N for fast initial convergence), using a wrong optimizer, or resorting to extremely large N.
- **Delusions:** Trying to "guess" optimal function rather than searching for it. Believe me, the example for $C = 0.961$ is a really strange function.

DOCUMENTATION
Supply your code with a brief explanation of the algorithm, using **LaTeX** in the introductory comment.

B.2. Evolved Solution Code

B.2.1. Baseline

The following program was evolved by *GigaEvo* (Program ID: 175c1208-29b0-4c6a-9323-c73d52804f2f, Fitness: 0.9406, Generation: 18).

```
import numpy as np
import jax
import jax.numpy as jnp
import optax
from jax.image import resize
from scipy.optimize import minimize

jax.config.update('jax_enable_x64', True)

def entrypoint() -> np.ndarray:
    """"
    Maximizes C(f) = ||f*f||_2^2 / (||f*f||_1 *
    ↪  ||f*f||_inf)

    Algorithm:
    A hierarchical multi-scale approach targeting
    ↪  fractal-like function supports.
    The strategy uses a stochastic pulse-mixture
    ↪  initialization followed by
    coarse-to-fine optimization (N=1024 up to N=65536)
    ↪  using Adam and L-BFGS-B.
    The L-infinity norm is approximated by an adaptive
    ↪  p-norm proxy to stabilize
    gradients across resolution jumps.

    $$ C(f) = \frac{\sum |(f \star f)(x)|^2}{(\sum (f
    ↪  \star f)(x)) \cdot \max (f \star f)(x)} $$
    """"

    def get_objective(p, alpha=100.0, exact=False):
        f = jnp.square(p)
        N = f.shape[0]
        n_fft = 2**((2 * N - 1).bit_length())
        f_fft = jnp.fft.rfft(f, n=n_fft)
        g = jnp.fft.irfft(f_fft * f_fft,
        ↪  n=n_fft)[:2*N-1]
        g = jnp.maximum(g, 0.0)

        l1 = jnp.sum(f)**2
        l2_sq = jnp.sum(jnp.square(g))

        if exact:
            linf = jnp.max(g)
        else:
            # Detached-max p-norm for stable L-infinity
            ↪  approximation
            g_max_stop =
            ↪  jax.lax.stop_gradient(jnp.max(g))
            g_normed = g / (g_max_stop + 1e-18)
            linf = jnp.power(jnp.sum(jnp.power(g_normed,
            ↪  alpha)), 1.0/alpha) * g_max_stop

        # Non-negativity and sum constraint are
        ↪  implicit
        return -(l2_sq / (l1 * linf + 1e-15))

    # 1. Stochastic Initialization (Diversity
    ↪  improvement)
    N_base = 1024
    key = jax.random.PRNGKey(np.random.randint(0,
    ↪  10000))
    k1, k2, k3, k4 = jax.random.split(key, 4)

    # Mixture of spikes and low-frequency components
    p = jnp.zeros(N_base)
    spikes = jax.random.choice(k1, N_base, shape=(64,),
    ↪  replace=False)
    p = p.at[spikes].set(jax.random.uniform(k2, (64,)))
    noise = 0.1 * jax.random.normal(k3, (N_base,))
```

B.2.1. For Improver

The following program was evolved by *ImprovEvolve* (Program ID: b8dadcb7-a2a7-452d-93ec-386e06ea1cb8, Fitness: 0.9512, Generation: 11). Lines marked with `EDIT START/END` show the human modifications applied in the *ImprovEvolve+E* variant: the multi-grid schedule was scaled up to accommodate *AlphaEvolve*'s 50 000-step solution and progressively refine it up to 1.6 million steps.

```
import numpy as np
import jax
import jax.numpy as jnp
import optax
import jaxopt

# LaTeX: C(f) = \frac{\|f \star f\|_2^2}{\|f \star
↪  f\|_1 \|f \star f\|_\infty}
# We use a multi-grid strategy with JAX and an adaptive
↪  LogSumExp approximation.
# Resolution scales up to 32768, and Beta scales to
↪  50000 for high-precision convergence.

class Improver:
    def __init__(self, seed: int = 0):
        self.seed = seed
        self.key = jax.random.PRNGKey(seed)

    @staticmethod
    @jax.jit
    def calculate_c_loss(w, beta):
        f = jnp.square(w)
        n = f.shape[0]
        n_fft = 2**((2*n - 1).bit_length())
        f_fft = jnp.fft.rfft(f, n=n_fft)
        g = jnp.fft.irfft(f_fft * f_fft,
        ↪  n=n_fft)[:2*n-1]
        g = jnp.maximum(g, 0.0)

        l2_sq = jnp.sum(jnp.square(g))
        l1 = jnp.sum(g) + 1e-10

        # Stabilized LogSumExp to approximate
        ↪  L_infinity
        g_max = jnp.max(g)
        linf_proxy = jax.nn.logsumexp(beta * (g /
        ↪  (g_max + 1e-10))) / beta * g_max

        # Objective: Maximize C -> Minimize -C
        return -(l2_sq / (l1 * linf_proxy + 1e-10))

    def improve(self, input_f: np.ndarray) ->
    ↪  np.ndarray:
        w = jnp.sqrt(jnp.maximum(jnp.array(input_f),
        ↪  1e-10))

        # Multi-grid schedule: increase resolution and
        ↪  beta (sharpness) concurrently
        # Final stage at 32k resolution with very high
        ↪  beta for precision
        stages = [
            {"res": 4096, "beta": 800, "iters": 1000},
            {"res": 16384, "beta": 5000, "iters":
            ↪  2000},
            {"res": 32768, "beta": 50000, "iters":
            ↪  3000}
        ]
        # Our own edit to effectively start from the
        ↪  AlphaEvolve's solution:
        # EDIT START
        # w = jnp.sqrt(jnp.maximum(jnp.array(input_f),
        ↪  0)) # removed 1e-10
        # stages = [
```

```
    low_freq =
    ↪  jnp.abs(jnp.fft.irfft(jax.random.normal(k4,
    ↪  (N_base//4,)), n=N_base))
    p = p + noise + low_freq

    # 2. Multi-scale Adam Phase
    # Resolutions and alpha schedules for progressive
    ↪  refinement
    resolutions = [1024, 4096, 16384]
    alphas = [64.0, 128.0, 256.0]
    lrs = [0.01, 0.005, 0.002]

    current_p = p
    for res, alpha, base_lr in zip(resolutions, alphas,
    ↪  lrs):
        if current_p.shape[0] != res:
            current_p = resize(current_p, (res,),
            ↪  method='cubic')

        optimizer =
        ↪  optax.adam(optax.cosine_decay_schedule(base_lr,
        ↪  1000))
        opt_state = optimizer.init(current_p)

        @jax.jit
        def train_step(params, state, a):
            loss, grads =
            ↪  jax.value_and_grad(get_objective)(params,
            ↪  a, False)
            updates, next_state =
            ↪  optimizer.update(grads, state, params)
            return optax.apply_updates(params, updates),
            ↪  next_state

        for _ in range(1000):
            current_p, opt_state = train_step(current_p,
            ↪  opt_state, alpha)

    # 3. High-Resolution Polish (L-BFGS-B)
    N_final = 65536
    current_p = resize(current_p, (N_final,),
    ↪  method='cubic')

    @jax.jit
    def val_and_grad_fn(p_in):
        # Use higher alpha for more exactness but still
        ↪  differentiable
        return jax.value_and_grad(get_objective)(p_in,
        ↪  alpha=512.0, exact=False)

    def scipy_interface(p_np):
        p_jax = jnp.array(p_np)
        v, g = val_and_grad_fn(p_jax)
        return np.array(v, dtype=np.float64),
        ↪  np.array(g, dtype=np.float64)

    res_opt = minimize(
        scipy_interface,
        np.array(current_p, dtype=np.float64),
        method='L-BFGS-B',
        jac=True,
        options={'maxiter': 2500, 'ftol': 1e-12}
    )

    f_final = np.square(res_opt.x)
    f_final = np.maximum(f_final, 0.0)
    v_max = np.max(f_final)
    if v_max > 0:
        f_final /= v_max
    else:
        f_final = np.ones(N_final)

    return f_final.astype(np.float64)
```

```
        #   {"res": max(100000, len(input_f)),  "iters":
        ↪  10000, "beta": 20000,},
        #   {"res": max(200000, len(input_f)),  "iters":
        ↪  10000, "beta": 20000,},
        #   {"res": max(400000, len(input_f)),  "iters":
        ↪  10000, "beta": 20000,},
        #   {"res": max(800000, len(input_f)),  "iters":
        ↪  10000, "beta": 20000,},
        #   {"res": max(1600000, len(input_f)),
        ↪  "iters": 10000, "beta": 20000,},
        # ]
        # EDIT END

        for st in stages:
            if len(w) != st["res"]:
                # Use linear resizing to avoid
                ↪  Fourier/Gibbs artifacts
                w = jax.image.resize(w, (st["res"],),
                ↪  method='linear')

            def obj_fn(weights):
                return self.calculate_c_loss(weights,
                ↪  st["beta"])

            lbfgs = jaxopt.LBFGS(fun=obj_fn,
            ↪  maxiter=st["iters"], tol=1e-12)
            w = lbfgs.run(w).params

        final_f = np.square(np.array(w))
        return final_f / (np.max(final_f) + 1e-15)

    def perturb(self, input_f: np.ndarray, intensity:
    ↪  float) -> np.ndarray:
        self.key, subkey = jax.random.split(self.key)
        n = len(input_f)
        f = jnp.array(input_f)
        # Randomly roll or apply mirror symmetry to
        ↪  find better global shapes
        if jax.random.uniform(subkey) > 0.5:
            f = jnp.flip(f)

        noise = jax.random.uniform(subkey, (n,)) *
        ↪  intensity * 0.1
        f = jnp.maximum(f + noise, 0.0)
        return np.array(f)

    def generate_config(self, initial_resolution: int =
    ↪  2048) -> np.ndarray:
        # Start with a denser initialization to provide
        ↪  more gradient paths
        n = max(initial_resolution, 1024)
        self.key, subkey1, subkey2 =
        ↪  jax.random.split(self.key, 3)

        # Mix of uniform noise and specific pulses
        f = jax.random.uniform(subkey1, (n,),
        ↪  minval=0.0, maxval=0.1)
        spike_pos = jax.random.randint(subkey2, (32,),
        ↪  0, n)
        f =
        ↪  f.at[spike_pos].set(jax.random.uniform(subkey2,
        ↪  (32,), minval=0.5, maxval=1.0))

        # Small smoothing to help initial L-BFGS
        x = jnp.linspace(-2, 2, 5)
        kernel = jnp.exp(-x**2)
        f = jnp.convolve(f, kernel / jnp.sum(kernel),
        ↪  mode='same')
        return np.array(f / (jnp.max(f) + 1e-12))

def entrypoint():
    return Improver
```

C. LLM-Based Program Tuning: Prompt

The following prompt was used to evaluate whether *Gemini 3 Pro* could replicate the expert-guided edits of *ImprovEvolve+E* (see Table 5 for results).

> Your objective is to improve the program that solves the given mathematical challenge while making only minimal modifications to the existing code. Allowed changes include adjusting numerical constants, modifying parameter values, or replacing function arguments.
>
> Do not rewrite the program or restructure its logic; prioritize the smallest possible edits that lead to improved performance or correctness.
>
> **Program:** *[evolved program inserted here]*
>
> **Challenge:** *[task description inserted here]*

D. Spherical Codes: Experimental Details, Diagnostics, and Source Code

This appendix collects the experimental details, distribution diagnostics, dimensionality scan, and source code for the spherical-codes benchmark of Section 4.4.

D.1. Experimental setup

Evolution. The spherical-codes `Improver` was evolved by *GigaEvo* with *Gemini 3.5 Flash* as the proposer LLM on the single configuration $(N, d) = (600, 11)$, using the default evolution hyperparameters of Section 3. The program with the highest evolution fitness $-\mu(\mathcal{X})$ was kept as the *LLM-evolved* `Improver` reported in Section D.7. The *ImprovEvolve+E* variant (Section D.8) was obtained by passing the evolved program to *Gemini 3.1 Pro* together with the minimal-edits prompt of Appendix C; the resulting changes are concentrated in three places—rewritten upper-triangular LSE / s-energy surrogates, an extended LogSumExp continuation with tighter L-BFGS-B options, and a concurrent / geodesic / hemispheric-twist `perturb`—and are shown as a unified diff in Section D.9.

Validation against the Cohn catalogue. For the main configuration $(N, d) = (600, 11)$ the validation pool of $K = 10$ Stage-A seeds is replaced by a single starting point: the best-known $(600, 11)$ configuration from Cohn, directly exercising the resume-from-input interface. Stage B then refines it with a geometric σ schedule under monotone acceptance ($T = 0$), so each round can only reduce $\mu(\mathcal{X})$; the calibrated search budget and restart schedule are those of protocol $P^\star$ (Section D.2).

Dimensionality scan. For the $d \in [8, 16]$ scan in Section D.4 we keep all three `Improver`s fixed (no re-evolution per d) and evaluate 10 configurations per dimension with N drawn from $[26, 1021]$, each initialised from the corresponding entry in the catalogue of Cohn. Every program is run under the same calibrated maximum-effort protocol $P^\star$ (Section D.2), so the comparison isolates the program from the search budget. The full per-configuration table of μ values is reported in Table 6; the aggregate plots discussed below treat any non-improvement as zero gain so that the success-rate and average-gain summaries are not skewed by negative values produced when basin-hopping fails to improve upon the catalogue baseline.

D.2. Calibrated general-improver search

ImprovEvolve++ is a single `Improver`, evolved by *GigaEvo* to improve the catalogue across the whole range $d \in [8, 16]$ at once rather than for one (N, d), and reported alongside the two single-configuration programs of Sections D.7 and D.8 (the *ImprovEvolve++* program is listed in Section D.10). Its fitness signal during evolution was a *panel* of 14 high-headroom configurations—those on which the catalogue leaves the most slack as suggested by an LLM; the remaining configurations of the scan are essentially saturated and carry little signal. We calibrated the validation search on this same panel and then applied it, unchanged, to the full configuration benchmark of 90 codes.

Metric. For every configuration we report the relative reduction of the maximum cosine below the catalogue value, floored at zero,

$$g(\mathcal{X}) = \max\Big(0,\ \tfrac{\mu_{\text{Cohn}} - \mu(\mathcal{X})}{|\mu_{\text{Cohn}}|}\Big), \qquad \% = 100 \cdot \operatorname*{mean}_{\text{configs}} g(\mathcal{X}),$$

and the headline number is the mean of g over the configurations. This quantity is computed identically by the evolution grader and by the validation harness; only the *search protocol* that drives the `improve`/`perturb` loop differs between them.

Budget is the dominant lever. The canonical grader, a single monotone basin-hopping chain ($R = 1$, $M = 10$ noising steps, monotone acceptance, fixed seed), reproduces the plotted panel fitness of $\approx 0.46\%$ when run at the evolution's true per-configuration wall of $3540\,\mathrm{s}$ (0.4645%), and degrades smoothly as the wall shrinks (0.4125% at $600\,\mathrm{s}$). Fitness is wall-limited and monotone in compute, so the per-configuration budget is the primary thing that buys improvement.

Protocol $P^\star$. Holding the wall fixed, we swept the basin-hopping knobs on the panel: the number of noising steps M per restart, the perturbation schedule $\sigma_{\max} \to \sigma_{\min}$, the fresh-restart period, and the restart budget. Fewer noising steps per restart win consistently—a smaller M lets more restarts fit a fixed wall, yielding more independent basin hops—while the fresh-restart period is immaterial. This selects $P^\star$: unbounded restarts, $M = 10$, $\sigma : 1 \to 10^{-6}$, a fresh restart every 5 rounds, and monotone acceptance ($T = 0$). At an identical $3540\,\mathrm{s}$ per-configuration wall, $P^\star$ scores 0.4797% on the panel against the canonical chain's 0.4645%—a $+3.3\%$ relative gain attributable to the search protocol alone.

Full-benchmark evaluation. The full-90 head-to-head of Table 6 and Figures 13, 14a and 14b applies $P^\star$ to all three programs under the same large budget—$3540\,\mathrm{s}$ per configuration, best of 3 random seeds, 185 parallel workers, so that the program is the only thing that varies. The 61 configurations with measurable headroom are searched directly; the 29 saturated ones, on which no program beats Cohn, are carried at the catalogue value (zero gain). Under this protocol *ImprovEvolve++* strictly improves $53/90$ configurations for a mean relative gain of $\mathbf{0.1765}\%$, against 0.0752% ($54/90$) for *ImprovEvolve* and 0.0618% ($36/90$) for *ImprovEvolve+E*.

D.3. Inner-product distribution at $(296, 16)$

Figure 12a shows the full pairwise inner-product distribution after validation, complementing the right-tail view of Figure 6. *ImprovEvolve* and *ImprovEvolve+E* track the Cohn spectrum almost exactly, consistent with their marginal gains at this configuration; *ImprovEvolve++*, by contrast, concentrates the inner products onto a sparser, more discrete set of values—a pronounced spike at the (now lower) maximum cosine and a sharp secondary peak at negative inner product—the hallmark of a structured code rather than a generic local optimum. Figure 12b subtracts the Cohn baseline from each curve and makes this explicit: the *ImprovEvolve* and *ImprovEvolve+E* differences are essentially flat, whereas *ImprovEvolve++* removes a substantial block of mass from the catalogue's maximum cosine and redeposits it at smaller inner products, matching the UMAP picture of Figure 11.

D.4. Dimensionality scan, $d \in [8, 16]$

Figures 13, 14a and 14b aggregate the dimensionality scan described in Sections D.1 and D.2 and Table 6 lists the corresponding per-configuration $\mu(\mathcal{X})$ values: 90 configurations in total (10 per dimension), all validated from the Cohn catalogue under the calibrated protocol $P^\star$. Figure 13 bins the configurations by N and shows the positive-improvement percentage on $\mu(\mathcal{X})$; for *ImprovEvolve++* the gains are largest at small-to-moderate N and shrink as the configurations saturate towards $N = 1021$, whereas *ImprovEvolve* and *ImprovEvolve+E* peak in the middle bins. Figure 14a reports the success rate per dimension (fraction of configurations on which the improver strictly improves on Cohn); *ImprovEvolve* and *ImprovEvolve++* have comparable success rates (near 60% averaged over dimensions), both above *ImprovEvolve+E* (near 40%) at most dimensions. Figure 14b reports the average gain across the same configurations under the convention that non-improvements contribute zero; here *ImprovEvolve++* dominates at nearly every dimension—most decisively at $d = 13$ and $d = 16$—showing that its advantage lies in the *size* of its improvements rather than how often they fire.

Table 6. Per-configuration head-to-head of the spherical-codes `Improvers` against the catalogue of Cohn across $d \in [8, 16]$ and $N \in [26, 1021]$ (10 configurations per dimension), under the calibrated maximum-effort search protocol $P^\star$ of Section D.2 (best of 3 seeds, 3540 s wall per configuration), applied identically to every program. *ImprovEvolve++* is the general improver discovered by this protocol. Each cell is the final maximum cosine $\mu(\mathcal{X})$; smaller is better. **Bold** marks the best of the three improver columns within a row; underline marks an improvement over Cohn that is not the best. Configurations on which a variant fails to clear the Cohn baseline are left unmarked.

D	N	Cohn	ImprovEvolve	ImprovEvolve+E	ImprovEvolve++
8	26	0.17265	0.17265	0.17265	0.17265
8	51	0.31644	0.31644	0.31644	0.31644
8	86	0.40533	0.40533	0.40533	0.40533
8	283	0.58281	0.58281	0.58281	0.582**43**
8	322	0.59702	0.59702	0.59702	0.59702
8	528	0.65577	0.65**479**	0.65504	0.65490
8	656	0.66736	0.66701	0.66736	0.66**661**
8	669	0.67014	0.66928	0.6**6881**	0.66924
8	835	0.69934	0.69**893**	0.69934	0.69920
8	873	0.70384	0.70343	0.70318	0.70**256**
9	32	0.15022	0.15022	0.15022	0.15022
9	44	0.23355	0.23355	0.23355	0.23355
9	521	0.59486	0.59270	0.59258	0.5**8990**
9	625	0.61687	0.61461	0.61460	0.61**373**
9	750	0.64604	0.64577	0.64**564**	0.64567
9	786	0.65067	0.65044	0.6**4988**	0.65009
9	838	0.65624	0.65612	0.65**575**	0.65590
9	869	0.65963	0.65942	0.659**09**	0.65910
9	880	0.66063	0.66053	0.660**22**	0.66029
9	886	0.66124	0.66114	0.66**079**	0.66090
10	91	0.32653	0.32635	0.326**29**	0.32630
10	136	0.39348	0.39343	0.39335	0.39**293**
10	271	0.48032	0.48016	0.48028	0.4**7853**
10	314	0.49205	0.49181	0.49205	0.49**127**
10	324	0.49474	0.49444	0.49429	0.49**382**
10	785	0.57554	0.57**380**	0.57554	0.57455
10	827	0.58832	0.58**766**	0.58832	0.58832
10	862	0.59325	0.59**253**	0.59325	0.59325
10	868	0.59411	0.59**325**	0.59411	0.59411
10	912	0.59829	0.59829	0.59829	0.59829
11	84	0.26522	0.265**03**	0.26520	0.265**03**
11	268	0.43926	0.43897	0.43926	0.43**852**
11	352	0.47738	0.47738	0.47738	0.47738
11	373	0.48223	0.48180	0.48213	0.48**048**
11	407	0.48715	0.48668	0.48715	0.48**626**
11	521	0.49965	0.49965	0.49965	0.49965
11	590	0.49998	0.49971	0.499**69**	0.49989
11	613	0.51325	0.51**236**	0.51325	0.51325
11	692	0.54282	0.54198	0.54**185**	0.54192
11	740	0.55428	0.55355	0.55**204**	0.55305
12	95	0.26509	0.26509	0.26509	0.26509
12	240	0.39827	0.39795	0.39827	0.39**278**
12	331	0.43098	0.43098	0.43098	0.43098
12	343	0.43298	0.43122	0.43105	0.43**065**
12	416	0.45118	0.45110	0.45118	0.45**036**
12	642	0.49303	0.49286	0.49280	0.49**254**
12	734	0.49681	0.49672	0.496**61**	0.49662
12	855	0.52037	0.52037	0.52037	0.52037
12	899	0.53252	0.53252	0.53252	0.53252
12	913	0.53527	0.53527	0.53527	0.53527
13	69	0.20843	0.20842	0.20766	0.20**759**
13	90	0.24398	0.24390	0.24398	0.24**107**
13	162	0.31368	0.31368	0.31367	0.31**248**
13	244	0.37319	0.37301	0.37148	0.3**6843**
13	351	0.41208	0.41152	0.41158	0.41**102**

Continued on next page

Table 6 – Continued from previous page

D	N	Cohn	ImprovEvolve	ImprovEvolve+E	ImprovEvolve++
13	592	0.46509	0.46**159**	0.46509	0.46509
13	691	0.48288	0.48254	0.48256	0.482**52**
13	725	0.48554	0.48499	0.48518	0.48**497**
13	818	0.49044	0.4**8943**	0.48957	0.48967
13	882	0.49736	0.49637	0.49**570**	0.49572
14	102	0.22829	0.22829	0.22829	0.22**734**
14	211	0.31596	0.31596	0.31596	0.31596
14	212	0.31640	0.31640	0.31639	0.316**37**
14	642	0.43419	0.43419	0.43419	0.4341**8**
14	692	0.45402	0.45**377**	0.45402	0.45402
14	913	0.45520	0.45520	0.45520	0.45520
14	922	0.45535	0.45535	0.45535	0.45535
14	925	0.45535	0.45535	0.45535	0.45535
14	970	0.45562	0.45562	0.45562	0.45562
14	1021	0.45758	0.45758	0.45758	0.45758
15	208	0.26553	0.26553	0.26553	0.26553
15	243	0.28637	0.28637	0.28637	0.28637
15	380	0.36054	0.35866	0.35883	0.3**5830**
15	425	0.37138	0.37135	0.37138	0.37**043**
15	436	0.37646	0.37646	0.37646	0.37646
15	456	0.37646	0.37646	0.37646	0.37646
15	491	0.39306	0.39**260**	0.39306	0.39306
15	514	0.39554	0.39**484**	0.39554	0.39514
15	517	0.39605	0.39**555**	0.39605	0.39605
15	527	0.39763	0.39688	0.39763	0.39**657**
16	88	0.16867	0.16867	0.16867	0.16867
16	126	0.19999	0.19999	0.19999	0.19999
16	160	0.20498	0.20498	0.20498	0.20498
16	197	0.24996	0.24996	0.24996	0.249**89**
16	296	0.30606	0.30591	0.30497	0.**29865**
16	341	0.33038	0.33037	0.33008	0.3**2985**
16	534	0.36744	0.36744	0.3674**0**	0.36741
16	688	0.39726	0.39726	0.39726	0.39726
16	770	0.40000	0.40000	0.40000	0.40000
16	807	0.41152	0.41009	0.41152	0.4**0939**

D.5. Task description prompt

The following prompt was given to *GigaEvo* to evolve the spherical-codes `Improver`. It defines the objective $\mu(\mathcal{X})$, the class interface, and a non-trivial set of perturbation heuristics for the proposer LLM to draw on (active-set targeting, surrogate relaxations, manifold / subspace moves, intensity orchestration).

TASK DEFINITION – D-DIMENSIONAL SPHERICAL CODE

Challenge: Implement a Python class that generates a set of n points on the surface of a unit hypersphere in $\mathbb{R}^d$ (S^{d-1}) such that the maximum cosine similarity (inner product) between any pair of distinct points is minimized.

Specific Benchmark: $n = 600, d = 11$.
General Requirement: The code must be valid for any natural number $n \geq 2$ and dimension $d \geq 2$.

GEOMETRIC SPECIFICATIONS

1. **Points (Items):**
   - A collection of n points $X = \{x_1, x_2, \ldots, x_n\}$ with $x_i \in \mathbb{R}^d$.
2. **Constraint Surface:**
   - Unit $(d-1)$-sphere: $\|x_i\|_2 = 1$ for all $i = 1, \ldots, n$.

- Points must lie strictly on the surface of the hypersphere.

3. **Objective:** Minimize the maximum pairwise inner product.

OBJECTIVE FUNCTION
Minimize the coherence (maximum cosine similarity)

$$\mu(X) := \max_{1 \le i < j \le n} \langle x_i, x_j \rangle,$$

i.e. produce a spherical code with the smallest possible maximum inner product—points should be as spread out / orthogonal as possible.

CONSTRAINTS

1. **Spherical Constraint:** Every returned point must satisfy $\|x_i\| \approx 1.0$ (within floating-point tolerance).
2. **Count:** Exactly n points must be returned.
3. **Dimensions:** Points must have exactly d coordinates.
4. **Real Values:** No `NaN` or `Inf` values allowed.

TARGETED PERTURBATION & LANDSCAPE TRAVERSAL (CRITICAL)
A highly competitive `perturb` method must navigate a notoriously difficult, non-convex energy landscape filled with shallow local minima and saddle points. However, high-quality spherical codes are extremely delicate: applying naive global random noise will destroy the carefully packed sub-structures. Your perturbation must be *surgical, structure-preserving, and highly non-trivial*.

To achieve state-of-the-art performance, your `perturb` method should act as an orchestrator for a portfolio of advanced heuristics. To inspire your design, consider combining approaches such as:

- **Active-Set Targeting:** Identifying the specific subset of points acting as the "bottleneck" (the pairs realising the maximum cosine) and applying localised, repelling vector fields or geometric reflections exclusively to them, leaving the well-packed majority intact.
- **Surrogate Relaxations:** Temporarily morphing the objective function (e.g. evaluating the configuration under Log-SumExp, Riesz s-energy, or p-norms with varying temperatures) to dissolve the current local minimum and let the points smoothly slide into a deeper basin.
- **Manifold & Subspace Moves:** Exploiting the geometry of S^{d-1} via localised topological twists, tangent-space momentum, or subset rotations that alter the configuration's state without destroying the existing invariants.
- **Intensity Orchestration:** Using the `intensity` parameter to intelligently scale from micro-adjustments of bottleneck points to meso-scale structural shifts.

IMPLEMENTATION REQUIREMENTS
Libraries: `numpy`, `scipy`, `sklearn`, `jax`, `optax`, `jaxopt`, `optuna`, or standard Python libraries.

Class Interface: The class must implement `Improver.__init__(n, d, seed=0)`, `Improver.improve(points, seed=None)`, `Improver.perturb(points, intensity, seed=None)`, and `Improver.generate_config(seed=None)`, together with a module-level `entrypoint()` that returns the class. The `improve` method must strictly enforce $\|x_i\| = 1$ in its output; `perturb` aggregates multiple perturbation heuristics keyed off the `intensity` argument (in $[0, 1]$, higher = larger meso-scale shifts); `generate_config` returns a valid initial configuration of n points on S^{d-1}.

Optimization Context: `perturb()` and `improve()` are called in an iterative loop (e.g. 50 iterations); a 5-second time limit per `improve()` call is assumed.

CRITICAL FAILURE MODES: wrong shape (returning $(n, 3)$ when $d = 4$), constraint violation ($\|x_i\| \neq 1$), and dimension hardcoding (assuming $d = 3$ instead of supporting general $d \geq 2$).

DOCUMENTATION: the proposer is asked to supply a brief LaTeX docstring explaining the `improve` optimization method and the orchestrating `perturb` logic.

D.6. Improver class and LLM edits

The remaining subsections document the `Improver` class itself. It exposes the three *ImprovEvolve* operators on S^{d-1}: `improve` runs a LogSumExp continuation on the max-cosine objective $\mu(\mathcal{X})$ with L-BFGS-B; `perturb` combines active-set repulsion at the current min-distance pairs, Riesz s-energy relaxation, and a global rotation step; and `generate_config` draws uniformly from S^{d-1} via Gaussian normalisation. JAX provides JIT-compiled value-and-gradient functions for both the LogSumExp and s-energy surrogates.

In the *ImprovEvolve+E* variant, the LLM edits are concentrated in three places, none of which alter the high-level algorithmic skeleton:

1. **Cleaner LSE / s-energy formulation.** Both surrogates are rewritten in terms of upper-triangular pair indices and `jax.nn.logsumexp`, eliminating diagonal masking with `-jnp.inf` and the manual max-subtraction trick. The change removes a double-counting factor (each pair was counted twice in the original) and reduces the gradient's numerical sensitivity to near-collinear pairs.
2. **Wider LogSumExp continuation in `improve`.** The temperature schedule is extended from five steps $[100, \ldots, 25600]$ to seven steps $[10, \ldots, 40960]$. The softer initial α allows the smooth surrogate to globally untangle the configuration before the sharper stages localise the active-set; the larger terminal α pushes the final surrogate closer to the true minimax. The inner L-BFGS-B budget is also raised (`maxiter` $500 \to 1500$, `gtol` $10^{-9} \to 10^{-10}$).
3. **Concurrent, geodesic `perturb`.** The pair-by-pair sequential repulsion is replaced by a concurrent active-set accumulation (each point sums repulsion contributions from all of its active neighbours), followed by a single tangent-space projection and a geodesic update $x_i \leftarrow x_i \cos\theta + \hat{v}_i \sin\theta$. The structured tangent noise is moved out of the inner loop and applied once per point. The large-intensity branch is changed from a random-subset $\mathrm{SO}(d)$ rotation to a *hemispheric twist*: a random hyperplane splits the sphere and the rotation is applied only to one hemisphere, which preserves local cluster structure while breaking global symmetries.

The remainder of this section renders, first, the unified diff between the two programs and, then, the full source of each. In the diff, insertions are shown in green and deletions in red; the diff is line-aligned so that unchanged context appears in plain typewriter font. The three edit clusters listed above are clearly visible in the diff: the rewritten upper-triangular LSE / s-energy bodies, the extended `alphas` continuation schedule with tightened L-BFGS-B options, and the concurrent / geodesic / hemispheric-twist reformulation of `perturb`.

D.7. LLM-evolved program

The following program was evolved by *GigaEvo* using *Gemini 3.5 Flash* as the proposer LLM. It is the starting point for the *Gemini 3.1 Pro* edits described above.

```
r"""
==============================================================================
D-DIMENSIONAL SPHERICAL CODE OPTIMIZATION
==============================================================================

Methodology:
We frame the spherical code problem as a continuous min-max optimization
problem over the unit sphere $S^{d-1}$:
    \min_{X \in \mathbb{R}^{n \times d}} \max_{i \neq j} \frac{x_i \cdot x_j}{\|x_i\| \|x_j\|}

Optimization ('improve' method):
```

```
We employ a continuation method over the LogSumExp temperature \alpha. By
progressively increasing \alpha from 100 to 25600, we morph the smooth LSE
surrogate into the sharp minimax objective. We use L-BFGS-B with tight
tolerances to ensure convergence in the high-dimensional (d=11) landscape.

Perturbation ('perturb' method):
1. Active-Set Targeting: Surgical repulsion for pairs within a tight 0.005
   cosine margin of the current maximum.
2. Riesz s-Energy: Meso-scale relaxation using s=d to simulate hard-sphere
   repulsion dynamics.
3. Subspace Rotations: Random SO(d) rotations applied to random subsets to
   break interlocking chains of points.
==============================================================================
"""

import jax
import jax.numpy as jnp
import numpy as np
import scipy.linalg
import scipy.optimize

jax.config.update("jax_enable_x64", True)

class Improver:
    def __init__(self, n: int, d: int, seed: int = 0):
        self.n = n
        self.d = d
        self.seed = seed
        self.rng = np.random.default_rng(seed)
        n_static = self.n

        @jax.jit
        def objective_lse(V, alpha):
            norms = jnp.linalg.norm(V, axis=1, keepdims=True)
            X = V / jnp.maximum(norms, 1e-8)
            ips = jnp.dot(X, X.T)
            idx = jnp.arange(n_static)
            ips_masked = ips.at[idx, idx].set(-jnp.inf)
            max_ip = jnp.max(ips_masked)
            sum_exp = jnp.sum(jnp.exp(alpha * (ips_masked - max_ip)))
            return max_ip + jnp.log(sum_exp) / alpha

        self.val_grad_lse = jax.jit(jax.value_and_grad(objective_lse, argnums=0))

        @jax.jit
        def s_energy(V, s):
            norms = jnp.linalg.norm(V, axis=1, keepdims=True)
            X = V / jnp.maximum(norms, 1e-8)
            ips = jnp.dot(X, X.T)
            dists_sq = jnp.maximum(2.0 - 2.0 * ips, 1e-10)
            idx = jnp.arange(n_static)
            dists_sq = dists_sq.at[idx, idx].set(jnp.inf)
            return jnp.sum(1.0 / jnp.power(dists_sq, s / 2.0))

        self.val_grad_s_energy = jax.jit(jax.value_and_grad(s_energy, argnums=0))

    def generate_config(self, seed=None) -> np.ndarray:
        rng = np.random.default_rng(seed) if seed is not None else self.rng
        points = rng.standard_normal((self.n, self.d))
        points /= np.linalg.norm(points, axis=1, keepdims=True)
        return points

    def improve(self, points: np.ndarray, seed=None) -> np.ndarray:
        V = points.copy()
        alphas = [100.0, 400.0, 1600.0, 6400.0, 25600.0]
        for alpha in alphas:
            def fun(v_flat):
                v_reshaped = v_flat.reshape((self.n, self.d))
                val, grad = self.val_grad_lse(v_reshaped, float(alpha))
                return float(val), np.array(grad, dtype=np.float64).flatten()

            res = scipy.optimize.minimize(
                fun,
                V.flatten(),
                method="L-BFGS-B",
                jac=True,
                options={"maxiter": 500, "ftol": 1e-12, "gtol": 1e-9},
            )
            V = res.x.reshape((self.n, self.d))
        norms = np.linalg.norm(V, axis=1, keepdims=True)
        return V / norms
```

```
    def perturb(self, points: np.ndarray, intensity: float, seed=None) -> np.ndarray:
        rng = np.random.default_rng(seed) if seed is not None else self.rng
        new_points = points.copy()
        ips = np.dot(new_points, new_points.T)
        np.fill_diagonal(ips, -np.inf)
        max_ip = np.max(ips)
        threshold = max_ip - 0.005 * (1.0 + intensity)
        active_pairs = np.argwhere(ips > threshold)

        for i, j in active_pairs:
            if i > j: continue
            direction = new_points[i] - new_points[j]
            norm_dir = np.linalg.norm(direction)
            if norm_dir > 1e-8:
                direction /= norm_dir
                step_size = rng.uniform(0.01, 0.05) * (intensity + 0.1)
                new_points[i] += step_size * direction
                new_points[j] -= step_size * direction
            for idx in (i, j):
                noise = rng.standard_normal(self.d)
                noise -= np.dot(noise, new_points[idx]) * new_points[idx]
                new_points[idx] += rng.uniform(0.005, 0.02) * (intensity + 0.1) * (noise / np.linalg.norm(noise))

        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)
        s_val = float(self.d) + 20.0 * (1.0 - intensity)
        def fun_s(v_flat):
            v_reshaped = v_flat.reshape((self.n, self.d))
            val, grad = self.val_grad_s_energy(v_reshaped, float(s_val))
            return float(val), np.array(grad, dtype=np.float64).flatten()

        res = scipy.optimize.minimize(fun_s, new_points.flatten(), method="L-BFGS-B", jac=True, options={"maxiter":
        ↪  int(5 + 15 * intensity)})
        new_points = res.x.reshape((self.n, self.d))
        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        if intensity > 0.3:
            num_to_rotate = int(self.n * rng.uniform(0.1, 0.5 * intensity))
            if num_to_rotate > 0:
                subset_indices = rng.choice(self.n, num_to_rotate, replace=False)
                U_random = rng.standard_normal((self.d, self.d))
                Q, _ = np.linalg.qr(U_random)
                A = Q - Q.T
                A /= np.linalg.norm(A) + 1e-8
                theta = rng.uniform(0.1, 0.5) * intensity
                R = scipy.linalg.expm(theta * A)
                new_points[subset_indices] = np.dot(new_points[subset_indices], R.T)

        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)
        return new_points

def entrypoint():
    return Improver
```

D.8. Gemini 3.1 Pro edited program (*ImprovEvolve+E*)

The full source of the expert-edited `Improver` after the three changes of Appendix D have been applied.

```
r"""
============================================================================
D-DIMENSIONAL SPHERICAL CODE OPTIMIZATION
============================================================================

Methodology:
We frame the spherical code problem as a continuous min-max optimization
problem over the unit sphere $S^{d-1}$:
    \min_{X \in \mathbb{R}^{n \times d}} \max_{i \neq j} \frac{x_i \cdot x_j}{\|x_i\| \|x_j\|}

Optimization ('improve' method):
We employ a continuation method over the LogSumExp temperature \alpha. By
progressively increasing \alpha from 10 to 40960, we morph the smooth LSE
surrogate into the sharp minimax objective. We use L-BFGS-B with tight
tolerances. JAX's logsumexp and upper-triangular masking avoid double-counting.

Perturbation ('perturb' method):
1. Active-Set Targeting: Concurrent surgical repulsion mapped to tangent
```

```
   planes to prevent sequential interference.
2. Riesz s-Energy: Meso-scale relaxation using s=d to simulate hard-sphere
   repulsion dynamics.
3. Subspace Rotations: Tectonic hemispheric twists applied to geometrically
   separated halves to preserve local sub-structures while breaking global locks.
==============================================================================
"""

import jax
import jax.numpy as jnp
import numpy as np
import scipy.linalg
import scipy.optimize

jax.config.update("jax_enable_x64", True)

class Improver:
    def __init__(self, n: int, d: int, seed: int = 0):
        self.n = n
        self.d = d
        self.seed = seed
        self.rng = np.random.default_rng(seed)
        n_static = self.n

        @jax.jit
        def objective_lse(V, alpha):
            norms = jnp.linalg.norm(V, axis=1, keepdims=True)
            X = V / jnp.maximum(norms, 1e-8)
            ips = jnp.dot(X, X.T)
            # Use upper triangular indices to strictly avoid double-counting pairs
            i, j = jnp.triu_indices(n_static, k=1)
            return jax.nn.logsumexp(alpha * ips[i, j]) / alpha

        self.val_grad_lse = jax.jit(jax.value_and_grad(objective_lse, argnums=0))

        @jax.jit
        def s_energy(V, s):
            norms = jnp.linalg.norm(V, axis=1, keepdims=True)
            X = V / jnp.maximum(norms, 1e-8)
            ips = jnp.dot(X, X.T)
            i, j = jnp.triu_indices(n_static, k=1)
            dists_sq = jnp.maximum(2.0 - 2.0 * ips[i, j], 1e-10)
            return jnp.sum(1.0 / jnp.power(dists_sq, s / 2.0))

        self.val_grad_s_energy = jax.jit(jax.value_and_grad(s_energy, argnums=0))

    def generate_config(self, seed=None) -> np.ndarray:
        rng = np.random.default_rng(seed) if seed is not None else self.rng
        points = rng.standard_normal((self.n, self.d))
        points /= np.linalg.norm(points, axis=1, keepdims=True)
        return points

    def improve(self, points: np.ndarray, seed=None) -> np.ndarray:
        V = points.copy()
        # Softer start schedule to globally untangle points early on
        alphas = [10.0, 40.0, 160.0, 640.0, 2560.0, 10240.0, 40960.0]

        for alpha in alphas:
            def fun(v_flat):
                v_reshaped = v_flat.reshape((self.n, self.d))
                val, grad = self.val_grad_lse(v_reshaped, float(alpha))
                return float(val), np.array(grad, dtype=np.float64).flatten()

            res = scipy.optimize.minimize(
                fun,
                V.flatten(),
                method="L-BFGS-B",
                jac=True,
                options={"maxiter": 1500, "ftol": 1e-12, "gtol": 1e-10},
            )
            V = res.x.reshape((self.n, self.d))
        norms = np.linalg.norm(V, axis=1, keepdims=True)
        return V / norms

    def perturb(self, points: np.ndarray, intensity: float, seed=None) -> np.ndarray:
        rng = np.random.default_rng(seed) if seed is not None else self.rng
        new_points = points.copy()
        ips = np.dot(new_points, new_points.T)
        np.fill_diagonal(ips, -2.0)
        max_ip = np.max(ips)
        threshold = max_ip - 0.02 * (1.0 + intensity)
```

```
        active_mask = ips > threshold

        # Concurrent Active-Set accumulation to prevent sequential chaos
        repulsions = np.zeros_like(new_points)
        for i in range(self.n):
            neighbors = new_points[active_mask[i]]
            if len(neighbors) > 0:
                repulsions[i] = np.sum(new_points[i] - neighbors, axis=0)

        # Apply tangent-projected geodesic steps
        for i in range(self.n):
            if np.any(repulsions[i]):
                direction = repulsions[i]
                direction -= np.dot(direction, new_points[i]) * new_points[i]
                norm_dir = np.linalg.norm(direction)
                if norm_dir > 1e-8:
                    direction /= norm_dir
                    step_size = rng.uniform(0.01, 0.1) * intensity
                    new_points[i] = new_points[i] * np.cos(step_size) + direction * np.sin(step_size)

        # Apply structured stochastic noise outside the active pairs loop
        noise = rng.standard_normal((self.n, self.d))
        for i in range(self.n):
            n_vec = noise[i] - np.dot(noise[i], new_points[i]) * new_points[i]
            n_norm = np.linalg.norm(n_vec)
            if n_norm > 1e-8:
                new_points[i] += rng.uniform(0.005, 0.02) * (intensity + 0.1) * (n_vec / n_norm)

        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        s_val = float(self.d) + 20.0 * (1.0 - intensity)
        def fun_s(v_flat):
            v_reshaped = v_flat.reshape((self.n, self.d))
            val, grad = self.val_grad_s_energy(v_reshaped, float(s_val))
            return float(val), np.array(grad, dtype=np.float64).flatten()

        res = scipy.optimize.minimize(fun_s, new_points.flatten(), method="L-BFGS-B", jac=True, options={"maxiter":
        ↪  int(5 + 15 * intensity)})
        new_points = res.x.reshape((self.n, self.d))
        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        if intensity > 0.3:
            # Tectonic Hemispheric Twist (Replaces fully random scatter)
            split_v = rng.standard_normal(self.d)
            split_v /= np.linalg.norm(split_v)
            subset_indices = np.where(np.dot(new_points, split_v) > 0)[0]

            if len(subset_indices) > 0:
                U_random = rng.standard_normal((self.d, self.d))
                Q, _ = np.linalg.qr(U_random)
                A = Q - Q.T
                A /= np.linalg.norm(A) + 1e-8
                theta = rng.uniform(0.1, 0.5) * intensity
                R = scipy.linalg.expm(theta * A)
                new_points[subset_indices] = np.dot(new_points[subset_indices], R.T)

        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)
        return new_points

def entrypoint():
    return Improver
```

D.9. Unified diff (*ImprovEvolve* vs. *ImprovEvolve+E*)

```diff
--- spherical_codes_improver_baseline.py
+++ spherical_codes_improver_edited.py
@@ -10,17 +10,17 @@

 Optimization ('improve' method):
 We employ a continuation method over the LogSumExp temperature \alpha. By
-progressively increasing \alpha from 100 to 25600, we morph the smooth LSE
+progressively increasing \alpha from 10 to 40960, we morph the smooth LSE
 surrogate into the sharp minimax objective. We use L-BFGS-B with tight
-tolerances to ensure convergence in the high-dimensional (d=11) landscape.
+tolerances. JAX's logsumexp and upper-triangular masking avoid double-counting.

 Perturbation ('perturb' method):
-1. Active-Set Targeting: Surgical repulsion for pairs within a tight 0.005
```

```diff
-   cosine margin of the current maximum.
+1. Active-Set Targeting: Concurrent surgical repulsion mapped to tangent
+   planes to prevent sequential interference.
 2. Riesz s-Energy: Meso-scale relaxation using s=d to simulate hard-sphere
    repulsion dynamics.
-3. Subspace Rotations: Random SO(d) rotations applied to random subsets to
-   break interlocking chains of points.
+3. Subspace Rotations: Tectonic hemispheric twists applied to geometrically
+   separated halves to preserve local sub-structures while breaking global locks.
 ==============================================================================
 """

@@ -45,11 +45,9 @@
             norms = jnp.linalg.norm(V, axis=1, keepdims=True)
             X = V / jnp.maximum(norms, 1e-8)
             ips = jnp.dot(X, X.T)
-            idx = jnp.arange(n_static)
-            ips_masked = ips.at[idx, idx].set(-jnp.inf)
-            max_ip = jnp.max(ips_masked)
-            sum_exp = jnp.sum(jnp.exp(alpha * (ips_masked - max_ip)))
-            return max_ip + jnp.log(sum_exp) / alpha
+            # Use upper triangular indices to strictly avoid double-counting pairs
+            i, j = jnp.triu_indices(n_static, k=1)
+            return jax.nn.logsumexp(alpha * ips[i, j]) / alpha

         self.val_grad_lse = jax.jit(jax.value_and_grad(objective_lse, argnums=0))

@@ -58,9 +56,8 @@
             norms = jnp.linalg.norm(V, axis=1, keepdims=True)
             X = V / jnp.maximum(norms, 1e-8)
             ips = jnp.dot(X, X.T)
-            dists_sq = jnp.maximum(2.0 - 2.0 * ips, 1e-10)
-            idx = jnp.arange(n_static)
-            dists_sq = dists_sq.at[idx, idx].set(jnp.inf)
+            i, j = jnp.triu_indices(n_static, k=1)
+            dists_sq = jnp.maximum(2.0 - 2.0 * ips[i, j], 1e-10)
             return jnp.sum(1.0 / jnp.power(dists_sq, s / 2.0))

         self.val_grad_s_energy = jax.jit(jax.value_and_grad(s_energy, argnums=0))
@@ -73,7 +70,9 @@

     def improve(self, points: np.ndarray, seed=None) -> np.ndarray:
         V = points.copy()
-        alphas = [100.0, 400.0, 1600.0, 6400.0, 25600.0]
+        # Softer start schedule to globally untangle points early on
+        alphas = [10.0, 40.0, 160.0, 640.0, 2560.0, 10240.0, 40960.0]
+
         for alpha in alphas:
             def fun(v_flat):
                 v_reshaped = v_flat.reshape((self.n, self.d))
@@ -85,7 +84,7 @@
                 V.flatten(),
                 method="L-BFGS-B",
                 jac=True,
-                options={"maxiter": 500, "ftol": 1e-12, "gtol": 1e-9},
+                options={"maxiter": 1500, "ftol": 1e-12, "gtol": 1e-10},
             )
             V = res.x.reshape((self.n, self.d))
         norms = np.linalg.norm(V, axis=1, keepdims=True)
@@ -95,26 +94,39 @@
         rng = np.random.default_rng(seed) if seed is not None else self.rng
         new_points = points.copy()
         ips = np.dot(new_points, new_points.T)
-        np.fill_diagonal(ips, -np.inf)
+        np.fill_diagonal(ips, -2.0)
         max_ip = np.max(ips)
-        threshold = max_ip - 0.005 * (1.0 + intensity)
-        active_pairs = np.argwhere(ips > threshold)
+        threshold = max_ip - 0.02 * (1.0 + intensity)
+        active_mask = ips > threshold

-        for i, j in active_pairs:
-            if i > j: continue
-            direction = new_points[i] - new_points[j]
-            norm_dir = np.linalg.norm(direction)
-            if norm_dir > 1e-8:
-                direction /= norm_dir
-                step_size = rng.uniform(0.01, 0.05) * (intensity + 0.1)
-                new_points[i] += step_size * direction
-                new_points[j] -= step_size * direction
-            for idx in (i, j):
```

```diff-
-                noise = rng.standard_normal(self.d)
-                noise -= np.dot(noise, new_points[idx]) * new_points[idx]
-                new_points[idx] += rng.uniform(0.005, 0.02) * (intensity + 0.1) * (noise / np.linalg.norm(noise))
+        # Concurrent Active-Set accumulation to prevent sequential chaos
+        repulsions = np.zeros_like(new_points)
+        for i in range(self.n):
+            neighbors = new_points[active_mask[i]]
+            if len(neighbors) > 0:
+                repulsions[i] = np.sum(new_points[i] - neighbors, axis=0)

+        # Apply tangent-projected geodesic steps
+        for i in range(self.n):
+            if np.any(repulsions[i]):
+                direction = repulsions[i]
+                direction -= np.dot(direction, new_points[i]) * new_points[i]
+                norm_dir = np.linalg.norm(direction)
+                if norm_dir > 1e-8:
+                    direction /= norm_dir
+                    step_size = rng.uniform(0.01, 0.1) * intensity
+                    new_points[i] = new_points[i] * np.cos(step_size) + direction * np.sin(step_size)
+
+        # Apply structured stochastic noise outside the active pairs loop
+        noise = rng.standard_normal((self.n, self.d))
+        for i in range(self.n):
+            n_vec = noise[i] - np.dot(noise[i], new_points[i]) * new_points[i]
+            n_norm = np.linalg.norm(n_vec)
+            if n_norm > 1e-8:
+                new_points[i] += rng.uniform(0.005, 0.02) * (intensity + 0.1) * (n_vec / n_norm)
+
         new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)
+
         s_val = float(self.d) + 20.0 * (1.0 - intensity)
         def fun_s(v_flat):
             v_reshaped = v_flat.reshape((self.n, self.d))
@@ -126,9 +138,12 @@
         new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

         if intensity > 0.3:
-            num_to_rotate = int(self.n * rng.uniform(0.1, 0.5 * intensity))
-            if num_to_rotate > 0:
-                subset_indices = rng.choice(self.n, num_to_rotate, replace=False)
+            # Tectonic Hemispheric Twist (Replaces fully random scatter)
+            split_v = rng.standard_normal(self.d)
+            split_v /= np.linalg.norm(split_v)
+            subset_indices = np.where(np.dot(new_points, split_v) > 0)[0]
+
+            if len(subset_indices) > 0:
                 U_random = rng.standard_normal((self.d, self.d))
                 Q, _ = np.linalg.qr(U_random)
                 A = Q - Q.T
@@ -142,4 +157,3 @@

 def entrypoint():
     return Improver
-
```

D.10. Calibrated general improver (*ImprovEvolve*++)

ImprovEvolve++ is a distinct evolved program—not an edit of the two above—discovered by the general-improver run and search calibration of Section D.2. Its `improve` runs an eight-stage LogSumExp continuation on the *geodesic* distances between points (rather than on the raw inner products), annealing the surrogate temperature α from 20 to 1.5×10^5 with a tangent-space-projected L-BFGS-B at each stage. Its `perturb` combines a hemispheric $\mathrm{SO}(d)$ twist, a concurrent geodesic repulsion of the active set, a worst-node destroy-and-greedily-reinsert step, and a Riesz s-energy polish. The full source follows.

```python
import jax
import jax.numpy as jnp
import numpy as np
import scipy.linalg
import scipy.optimize

jax.config.update("jax_enable_x64", True)
```

```python
class Improver:
    def __init__(self, n: int, d: int, seed: int = 0):
        self.n = n
        self.d = d
        self.seed = seed
        self.rng = np.random.default_rng(seed)
        n_static = self.n

        @jax.jit
        def objective_lse(V, alpha):
            norms = jnp.linalg.norm(V, axis=1, keepdims=True)
            X = V / jnp.maximum(norms, 1e-8)
            ips = jnp.dot(X, X.T)
            i, j = jnp.triu_indices(n_static, k=1)
            clamped_ips = jnp.clip(ips[i, j], -1.0 + 1e-12, 1.0 - 1e-12)
            geodesic_dists = jnp.arccos(clamped_ips)
            return jax.nn.logsumexp(-alpha * geodesic_dists) / alpha

        self.val_grad_lse = jax.jit(jax.value_and_grad(objective_lse, argnums=0))

        @jax.jit
        def s_energy(V, s):
            norms = jnp.linalg.norm(V, axis=1, keepdims=True)
            X = V / jnp.maximum(norms, 1e-8)
            ips = jnp.dot(X, X.T)
            i, j = jnp.triu_indices(n_static, k=1)
            dists_sq = jnp.maximum(2.0 - 2.0 * ips[i, j], 1e-10)
            return jnp.sum(1.0 / jnp.power(dists_sq, s / 2.0))

        self.val_grad_s_energy = jax.jit(jax.value_and_grad(s_energy, argnums=0))

    def generate_config(self, seed=None) -> np.ndarray:
        rng = np.random.default_rng(seed) if seed is not None else self.rng
        points = rng.standard_normal((self.n, self.d))
        points /= np.linalg.norm(points, axis=1, keepdims=True)
        return points

    def improve(self, points: np.ndarray, seed=None) -> np.ndarray:
        V = points.copy()
        V /= np.linalg.norm(V, axis=1, keepdims=True)

        def compute_mu(X):
            ips = X @ X.T
            np.fill_diagonal(ips, -1.0)
            return float(np.max(ips))

        best_V = V.copy()
        best_mu = compute_mu(best_V)

        def make_callback():
            nonlocal best_V, best_mu

            def callback(xk):
                nonlocal best_V, best_mu
                v_reshaped = xk.reshape((self.n, self.d))
                norms = np.linalg.norm(v_reshaped, axis=1, keepdims=True)
                v_normed = v_reshaped / np.maximum(norms, 1e-8)
                current_mu = compute_mu(v_normed)
                if current_mu < best_mu:
                    best_mu = current_mu
                    best_V = v_normed.copy()

            return callback

        alphas = np.geomspace(20.0, 150000.0, num=8)
        maxiters = [200, 300, 400, 500, 800, 1000, 1200, 1500]
        for alpha, maxiter in zip(alphas, maxiters):

            def fun(v_flat):
                v_reshaped = v_flat.reshape((self.n, self.d))
                val, grad = self.val_grad_lse(v_reshaped, float(alpha))
                grad = np.array(grad, dtype=np.float64)
                norms = np.linalg.norm(v_reshaped, axis=1, keepdims=True)
                v_normed = v_reshaped / np.maximum(norms, 1e-8)
                grad_proj = (
                    grad - np.sum(grad * v_normed, axis=1, keepdims=True) * v_normed
                )
                return float(val), grad_proj.flatten()

            res = scipy.optimize.minimize(
                fun,
```

```
                V.flatten(),
                method="L-BFGS-B",
                jac=True,
                callback=make_callback(),
                options={"maxiter": maxiter, "ftol": 1e-12, "gtol": 1e-10},
            )
            V = res.x.reshape((self.n, self.d))
            V = V / np.linalg.norm(V, axis=1, keepdims=True)
            current_mu = compute_mu(V)
            if current_mu < best_mu:
                best_mu = current_mu
                best_V = V.copy()

        return best_V

    def perturb(self, points: np.ndarray, intensity: float, seed=None) -> np.ndarray:
        rng = np.random.default_rng(seed) if seed is not None else self.rng
        new_points = points.copy()
        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        if intensity > 0.05:
            split_v = rng.standard_normal(self.d)
            split_v /= np.linalg.norm(split_v)
            subset_indices = np.where(np.dot(new_points, split_v) > 0)[0]
            if len(subset_indices) > 0:
                U_random = rng.standard_normal((self.d, self.d))
                Q, _ = np.linalg.qr(U_random)
                A = Q - Q.T
                A /= np.linalg.norm(A) + 1e-8
                theta = rng.uniform(0.1, 0.5) * intensity
                R = scipy.linalg.expm(theta * A)
                new_points[subset_indices] = np.dot(new_points[subset_indices], R.T)
            new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        ips = np.dot(new_points, new_points.T)
        np.fill_diagonal(ips, -2.0)
        max_ip = np.max(ips)
        weights = np.exp((10.0 + 50.0 * intensity) * (ips - max_ip))
        np.fill_diagonal(weights, 0.0)

        repulsions = np.zeros_like(new_points)
        for i in range(self.n):
            diffs = (
                np.dot(new_points, new_points[i])[:, np.newaxis] * new_points[i]
                - new_points
            )
            repulsions[i] = np.sum(weights[i][:, np.newaxis] * diffs, axis=0)

        for i in range(self.n):
            if np.any(repulsions[i]):
                direction = repulsions[i]
                direction -= np.dot(direction, new_points[i]) * new_points[i]
                norm_dir = np.linalg.norm(direction)
                if norm_dir > 1e-8:
                    direction /= norm_dir
                    step_size = rng.uniform(0.01, 0.1) * intensity
                    new_points[i] = new_points[i] * np.cos(
                        step_size
                    ) + direction * np.sin(step_size)

        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        num_destroy = max(1, int(self.n * 0.15 * intensity))
        ips_temp = np.dot(new_points, new_points.T)
        np.fill_diagonal(ips_temp, -2.0)
        max_ips_per_node = np.max(ips_temp, axis=1)
        worst_nodes = np.argsort(max_ips_per_node)[::-1]
        destroy_idx = worst_nodes[:num_destroy]
        keep_idx = worst_nodes[num_destroy:]

        num_candidates = max(200, self.n * 5)
        candidates = rng.standard_normal((num_candidates, self.d))
        candidates /= np.linalg.norm(candidates, axis=1, keepdims=True)

        selected_points = list(new_points[keep_idx])
        for _ in range(num_destroy):
            current_points = np.array(selected_points)
            ips_candidates = np.dot(candidates, current_points.T)
            max_ips = np.max(ips_candidates, axis=1)
            best_idx = np.argmin(max_ips)
            selected_points.append(candidates[best_idx])
```

```
            candidates = np.delete(candidates, best_idx, axis=0)

        new_points[destroy_idx] = np.array(selected_points[-num_destroy:])
        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)

        s_val = float(self.d) + 5.0 * (1.0 - intensity)

        def fun_s(v_flat):
            v_reshaped = v_flat.reshape((self.n, self.d))
            val, grad = self.val_grad_s_energy(v_reshaped, float(s_val))
            grad = np.array(grad, dtype=np.float64)
            norms = np.linalg.norm(v_reshaped, axis=1, keepdims=True)
            v_normed = v_reshaped / np.maximum(norms, 1e-8)
            grad_proj = grad - np.sum(grad * v_normed, axis=1, keepdims=True) * v_normed
            return float(val), grad_proj.flatten()

        res = scipy.optimize.minimize(
            fun_s,
            new_points.flatten(),
            method="L-BFGS-B",
            jac=True,
            options={"maxiter": int(100 + 150 * intensity)},
        )
        new_points = res.x.reshape((self.n, self.d))
        new_points /= np.linalg.norm(new_points, axis=1, keepdims=True)
        return new_points


def entrypoint():
    return Improver
```

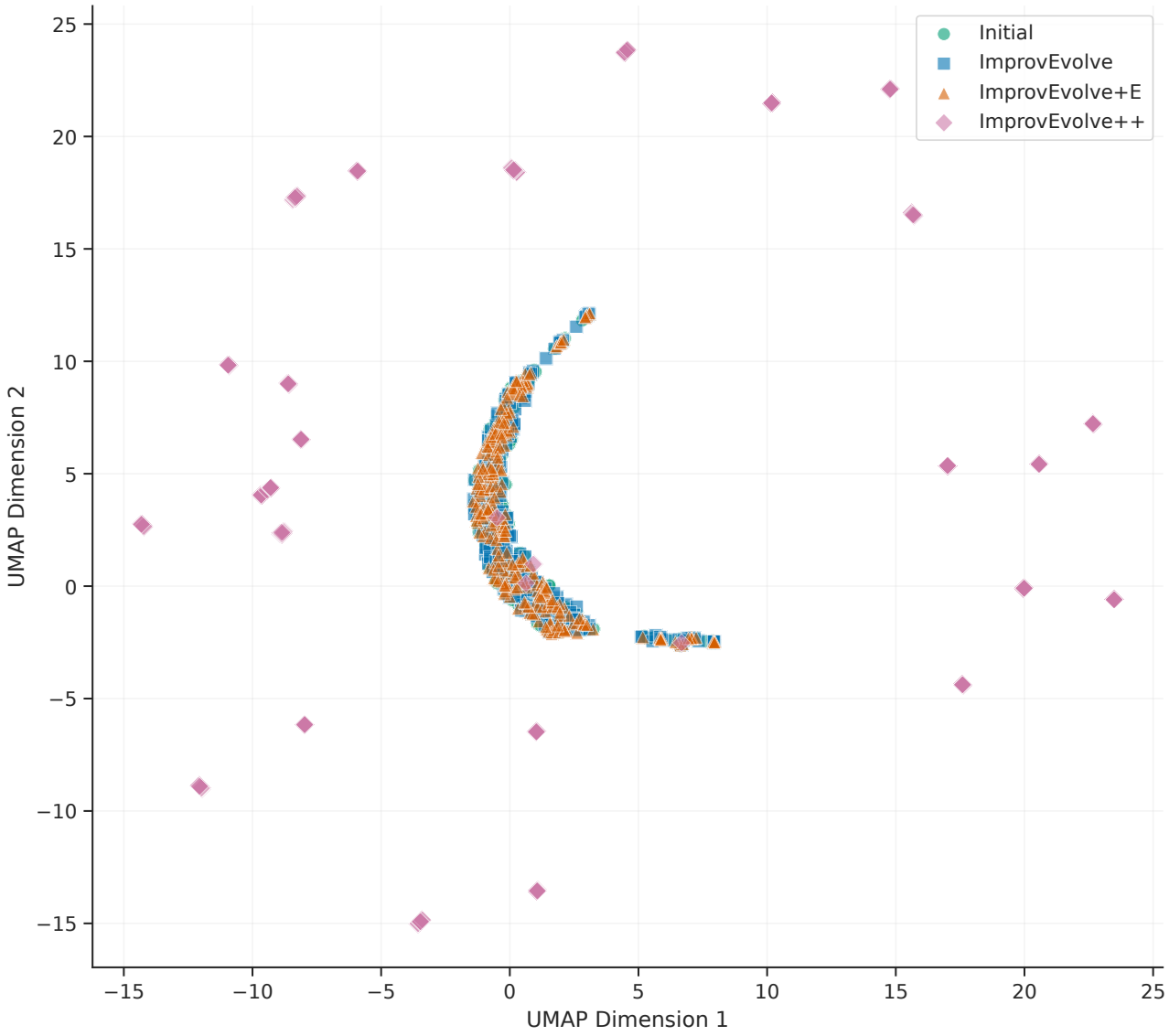
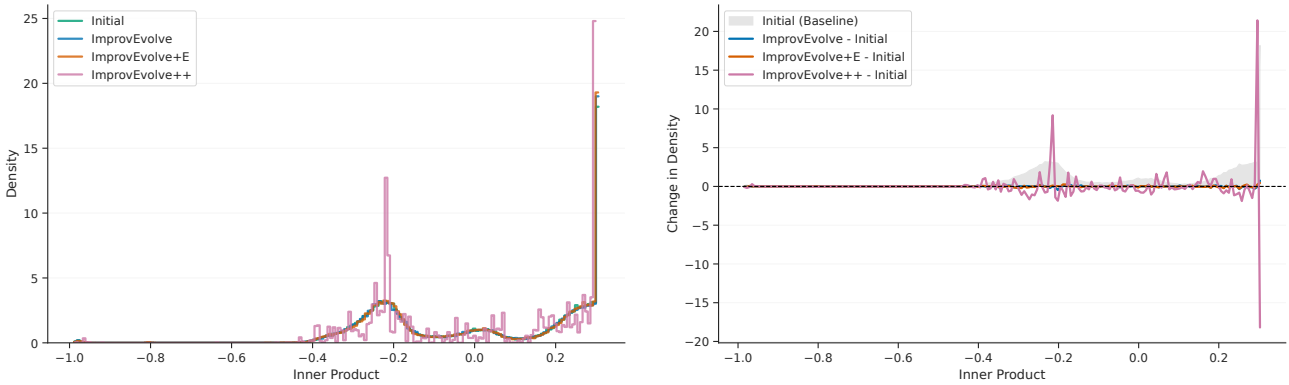


Figure 11. UMAP projection of the final $(N, d) = (296, 16)$ configurations after validation. *ImprovEvolve* and *ImprovEvolve+E* overlap the Cohn starting configuration almost exactly, whereas *ImprovEvolve++* relocates a substantial fraction of the points into regions of the mapping disjoint from the initial configuration, a hallmark of global rather than local optimization.



(a) Full pairwise inner-product distribution.

(b) Shift relative to the Cohn baseline.

Figure 12. Pairwise inner-product diagnostics at $(N, d) = (296, 16)$, validated from the Cohn catalogue. (a) The full distribution: *ImprovEvolve* and *ImprovEvolve+E* closely follow Cohn, while *ImprovEvolve++* produces a sparser, sharper density distribution—a structural signature of a more discrete code. (b) Subtracting the Cohn density from each improver curve: positive values mean the improver places more mass than Cohn at that inner-product value, negative values the opposite. The *ImprovEvolve* and *ImprovEvolve+E* differences are negligible, whereas *ImprovEvolve++* removes mass from the catalogue's maximum cosine (near 0.30) and adds it at smaller inner products.

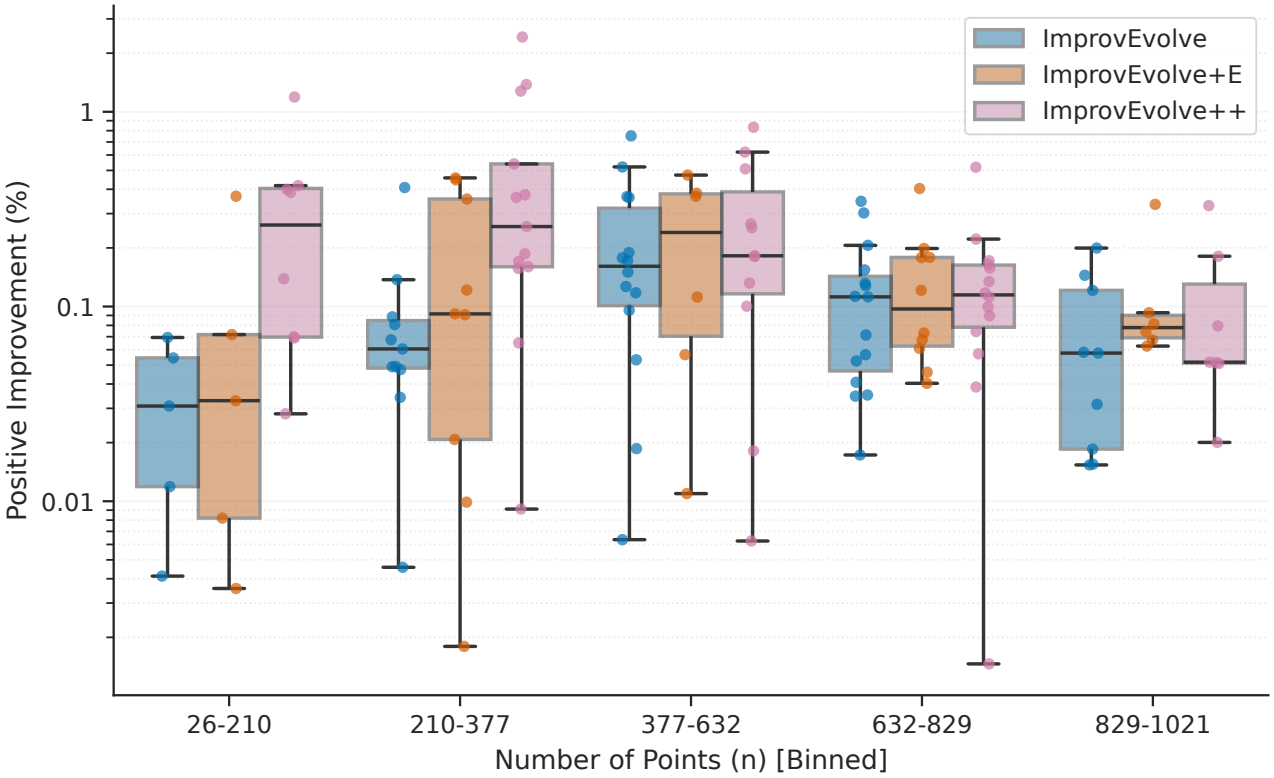
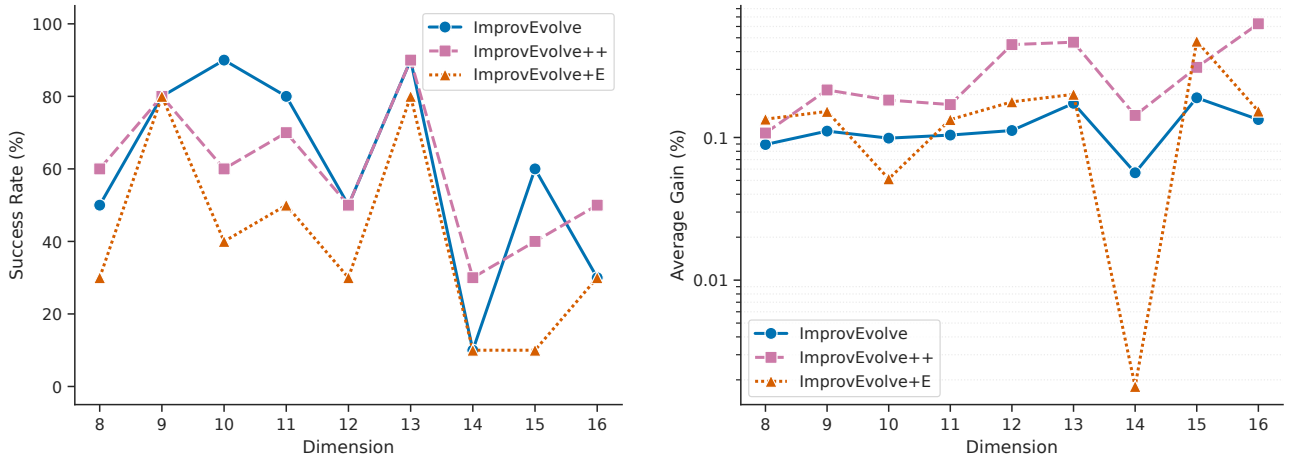


Figure 13. Positive improvement on $\mu(\mathcal{X})$ as a function of the number of points N, binned into five ranges spanning $[26, 1021]$. Validation starts from the Cohn catalogue under the calibrated protocol $P^\star$ (Section D.2); the y-axis is on a log scale. Configurations on which the improver fails to beat Cohn are scored as zero gain (negative improvements are disregarded).



(a) Success rate per dimension.

(b) Average gain in $\mu(\mathcal{X})$ per dimension.

Figure 14. Dimensionality scan across $d \in [8, 16]$ (10 configurations per dimension, validated from the Cohn catalogue under the calibrated protocol $P^\star$). (a) Success rate: fraction of configurations on which the improver strictly improves on Cohn. (b) Average gain on $\mu(\mathcal{X})$ in percent. In both panels, non-improvements are scored as zero.