CURIO: CURIOSITY-DRIVEN TEST-TIME LEARNING FOR OPEN-ENDED DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Open-ended discovery requires learning from repeated attempts while continuing to explore directions whose value is not yet apparent. Search with a frozen large language model (LLM) can reuse previous solutions in context, but cannot update the model from its successes and failures on the test problem. Reinforcement learning (RL) enables such adaptation; however, strongly favoring high-reward trajectories may suppress low-reward yet potentially promising directions too early. We introduce CURIO, a curiosity-driven test-time learning framework that complements task feedback with an Intrinsic Curiosity World Model (ICWM). The ICWM learns transitions in the policy's hidden-state representation and supplies prediction-error bonuses at sampled tokens outside the policy's top-k choices. Epoch normalization and an annealed weight regulate their contribution to the policy update. On six mathematical discovery tasks and single-cell denoising with Qwen3 backbones from 8B to 235B, three-run means improve over a matched task-only RL control on five mathematical objectives, match the best reported performance on Circle Packing, and improve denoising Score and mean squared error (MSE) on both held-out corpora at every tested scale. Relative gains reach 18.3% on Hadamard and 10.8% on denoising Score. Code-diversity measurements show greater structural variation among generated programs, supporting curiosity as a complementary exploration signal for learning in open-ended discovery.

1 INTRODUCTION

Open-ended scientific and algorithmic discovery seeks progressively better solutions rather than a single predetermined answer. LLMs can participate in this process by proposing programs, receiving evaluator feedback, and refining their attempts (Romera-Paredes et al., 2024; Novikov et al., 2025). This approach spans mathematical construction and broader AI R&D tasks (Tao, 2026; Rank et al., 2026; Wang et al., 2026; Wu et al., 2026). Progress depends both on exploiting successful attempts and on exploring directions whose eventual value may not be reflected in their current reward.

Existing search-based auto-research methods repeatedly sample from a frozen LLM, organizing previous attempts through prompts, archives, and populations (Novikov et al., 2025; Lange et al., 2026; Ye et al., 2026). These mechanisms can improve the context supplied to the model, but the model's parameters cannot learn from its successes and failures on the test problem. RL-based methods instead turn discovery into test-time RL, updating the proposing policy with experience collected from that same problem (Wang et al., 2025; Yuksekgonul et al., 2026). This enables the model to internalize useful patterns as discovery proceeds.

Learning from task rewards nevertheless creates an exploration challenge. Updates that strongly prioritize high-reward trajectories can concentrate the policy around currently successful constructions. A direction with low current reward may still admit valuable refinements, yet receive too little future sampling to reveal that potential. Thus, improved exploitation can suppress promising alternatives prematurely. Our central question is: *how can a policy learn from task feedback while retaining an incentive to explore underexplored directions?*

Task reward alone may provide insufficient incentive to revisit underexplored directions: it assesses the quality of a completed candidate rather than the potential of further exploration. Increasing sampling temperature broadens sampling, while entropy regularization encourages uncertainty without directly

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

measuring learned transition predictability. We therefore seek an exploration signal learned from the policy's own experience and applied selectively during generation.

This requires resolving two linked questions. First, *what experience should drive the exploration signal?* Surface differences between long code sequences need not imply different solution strategies. Policy hidden states instead provide a representation in which a learned model can assess transition predictability, although high prediction error does not by itself establish semantic novelty or future task improvement. Second, *where should the signal affect learning?* Rewarding surprise at every token can distract from task progress. A selective rule and a controlled reward scale are needed to combine exploration with task-driven improvement.

	Erdos		Circle Packing	
Training stage	CURIO	TTT-Discover	CURIO	TTT-Discover
Early (0–12)	329	290	326	294
Mid (13–25)	402	323	280	225
Late (26–38)	285	206	219	158
Final (39–49)	411	331	304	250

Figure 1: **Motivating exploration beyond task rewards.** On Erdős and Circle Packing, CURIO has higher code diversity than task-only RL (TTT-Discover) at every displayed stage (metric: Appendix F).

We propose CURIO, a curiosity-driven test-time learning framework for open-ended discovery. Within the RL discovery loop, an ICWM learns token-level transitions from policy hidden-state features and sampled-token embeddings. Its cached prediction residuals supply an intrinsic signal, normalized over each epoch and retained only when a sampled token lies outside the policy's top-k choices. An annealed weight combines this bonus with the task coefficient before the policy update. The ICWM models the policy's representation dynamics; the task evaluator continues to determine candidate quality. This supplies a complementary training signal without assuming that every surprising transition is useful or that every low-reward trajectory should be retained. Figure 1 illustrates the resulting comparison: CURIO exhibits higher code-level structural diversity than the task-only RL baseline at every displayed stage on both tasks. This diagnostic does not by itself establish semantic diversity or a causal mechanism.

We evaluate CURIO on six mathematical discovery tasks and single-cell denoising on the held-out peripheral blood mononuclear cell (PBMC) and Tabula corpora, with Qwen3 backbones from 8B to 235B. Relative to a matched task-only RL control, three-run means improve on five mathematical objectives and match the best reported performance on Circle Packing; denoising Score and MSE improve on both corpora at every tested scale. Relative improvements reach 18.3% on 235B Hadamard and 10.8% on 8B Tabula Score. Figure 3 extends the introductory code-diversity comparison to all six discovery trajectories. Ablations examine curiosity strength, predictor architecture, and representation coupling. Together, these results evaluate the benefit of adding a learned exploration signal to reward-driven adaptation, while code diversity and prediction error remain distinct from semantic novelty.

2 Preliminaries

2.1 Open-Ended Discovery

A discovery problem specifies a description d, candidate solutions, and an evaluator (Novikov et al., 2025; Yuksekgonul et al., 2026). The aim is to find the best feasible solution within a finite budget; improving on a prior best-known value constitutes a new record. An archive $\mathcal{H}$ stores evaluated candidates and their programs. Given a parent $p \in \mathcal{H}$, the prompt $x = x(d, p)$ conditions a policy π_θ, parameterized by θ, to produce a completion y, containing code and optional reasoning. Parsing and executing the code yields a new candidate for evaluation and possible archive insertion. We write $R(x, y)$ for its scalar training reward after feasibility checks and the task-specific transformation in Appendix D; larger R is better, even when the raw objective is minimized. The candidate p and completion y describe the discovery loop; s_t and a_t are reserved for the token-level ICWM quantities defined in §3.2.

2.2 RL for Discovery

RL-based discovery learns from evaluator feedback on the current problem by updating the proposing policy (Wang et al., 2025; Yuksekgonul et al., 2026). To emphasize high-reward candidates, a

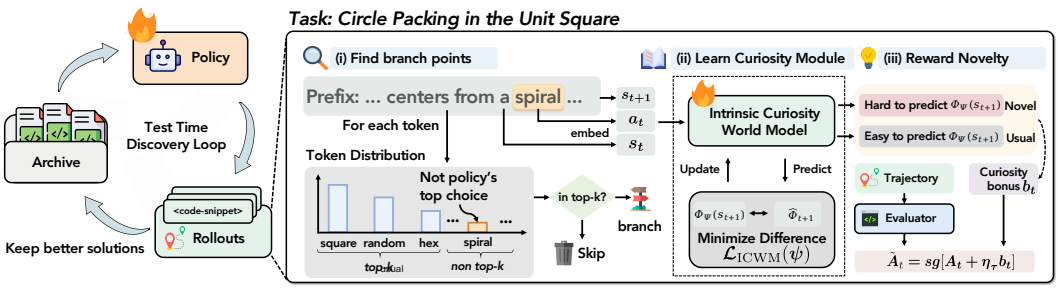


Figure 2: **Overview of CURIO.** A policy expands archive parents into rollouts. The ICWM learns token-level transitions in encoded policy hidden states, and its prediction residual informs the curiosity bonus. The signal is retained at sampled tokens outside the policy's top-k choices and combined with the task coefficient. The states s_t, s_{t+1} and action a_t are token-level features and embeddings. The diagram is schematic; Equations (5)–(12) specify the representations, normalization, and masking.

discovery-oriented objective uses exponential reward weighting (Yuksekgonul et al., 2026):

$$J_\beta(\theta; x) = \log \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[e^{\beta R(x,y)}], \qquad \beta > 0. \tag{1}$$

Here x is an archive-conditioned prompt, $R(x, y)$ is the evaluator reward, and β controls the emphasis on high rewards. Larger β more strongly favors high-reward completions, potentially reducing exploration of currently low-reward but promising directions.

3 Curiosity-Driven Test-Time Learning

CURIO follows the RL discovery loop in §2.2 and augments task-driven learning with a token-local curiosity signal. Figure 2 summarizes the three components: select non-top-k sampled tokens, learn an ICWM on policy representation transitions, and combine its prediction-error bonus with task feedback. We define the task-interface notation in §3.1, the ICWM and bonus in §3.2, and the joint update in §3.3. Algorithm 1 in Appendix B provides the complete procedure.

Overview. In the diagram, Φ_ψ and $\widehat{\Phi}_{t+1}$ correspond to ϕ_ψ and $\hat{\phi}_{t+1}$ below, and the displayed η_t means the epoch weight η_τ. "Branch points" denotes the non-top-k heuristic; "novel" and "usual" denote relative prediction difficulty, not established semantic novelty. "Skip" omits the curiosity bonus, not task learning or ICWM fitting. The bonus b_t includes epoch normalization and masking; it is not the raw residual.

3.1 Task Feedback and Token Notation

Notation and token indexing. At epoch τ, θ_{old} denotes the policy snapshot used for generation, and θ_0 denotes the initial reference policy. Only adapter parameters are updated. A prompt x has length $P \geqslant 1$, its completion $y = (y_1, \ldots, y_T)$ has length T, and $z = (x, y)$ is indexed from zero. Thus y_t occupies position $P + t - 1$ and has conditioning prefix $z_{<P+t-1}$. The mask $m_t \in \{0, 1\}$ is one for sampled, trainable completion tokens and zero for inserted forced-suffix tokens. Prompt and padding tokens are excluded from all completion-level sums. Let $\mathcal{D}_\tau$ be the epoch's training rollouts after filtering constant-reward groups, and let $M_\tau = \sum_{y\in\mathcal{D}_\tau} \sum_{t=1}^{T_y} m_t$. Here T_y is the length of rollout y; each rollout retains its associated prompt x. Sums over t run over its completion positions, and rollout indices on token quantities are suppressed. No training update is taken when $M_\tau = 0$.

Sampled task coefficients. We instantiate the exponential weighting of §2.2 using the following construction, shared with the matched task-only control. The task coefficient supplies reward-driven learning; the ICWM adds a separate exploration signal. For a group of $G \geqslant 2$ rollouts from one parent, sampled from $\pi_{\theta_{\text{old}}}(\cdot \mid x)$, with rewards $R_i = R(x, y^{(i)})$ for $i = 1, \ldots, G$, the sampled task coefficient is

$$A_i^{\text{task}} = \frac{e^{\beta R_i}}{\frac{1}{G-1}\sum_{j\neq i} e^{\beta R_j}} - 1. \tag{2}$$

The group-specific β is treated as fixed during the policy update. Its calibration targets a Kullback–Leibler (KL) divergence of $\delta = \log 2$ between $\text{softmax}(\beta(R_1, \ldots, R_G))$ and the uniform distribution

over the G rollouts when this target is attainable (Appendix A). Equation (2) is a finite-sample ratio estimator, not an unbiased estimate of the population-normalized exponential reward weight minus one. Only A_i^{task}, before the token-local KL correction below, is shared by every token of rollout i.

Token-local reference correction. For a token in rollout i, define the sampled log-ratio to the reference policy and its epoch mean:

$$d_t = \log \pi_{\theta_{\text{old}}}(y_t \mid z_{<P+t-1}) - \log \pi_{\theta_0}(y_t \mid z_{<P+t-1}), \tag{3}$$

$$\bar{d} = \frac{1}{M_\tau} \sum_{y \in \mathcal{D}_\tau} \sum_t m_t d_t, \qquad A_t = A_i^{\text{task}} + \lambda m_t (\bar{d} - d_t). \tag{4}$$

Here, the reference-correction coefficient $\lambda \geqslant 0$ controls a centered, token-local correction toward the reference policy. This is the coefficient used by the training procedure; it is not a sequence-level log-ratio broadcast to every token. Set $d_t = 0$ at inserted tokens, whose $m_t = 0$ prevents any contribution. All sampled coefficients and reference statistics are held fixed during the policy update. This sampled, empirically centered correction should not be identified with the exact gradient of a sequence-level KL-regularized population objective.

3.2 Intrinsic Curiosity World Model

The ICWM learns token-level transitions to supply an intrinsic exploration signal. It consists of a feature encoder ϕ_ψ and a forward predictor f_ψ operating on actor hidden states and token embeddings. It models transitions within the policy's representation space, not the external scientific environment or the evaluator. Its prediction residual is converted into a token-local bonus through epoch normalization and selective masking.

ICWM states and actions. Let d_h be the backbone hidden dimension and $h_j \in \mathbb{R}^{d_h}$ be the last-layer hidden state at position j from a no-gradient forward pass of $\pi_{\theta_{\text{old}}}$. For completion step t, define

$$s_t = h_{P+t-2}, \qquad a_t = \text{Emb}(y_t), \qquad s_{t+1} = h_{P+t-1}. \tag{5}$$

Here Emb is the frozen backbone token-embedding lookup, so $a_t \in \mathbb{R}^{d_h}$. In particular, $s_1 = h_{P-1}$ is the final prompt representation. The symbols s_t, a_t refer to token-level features and embeddings, whereas p denotes an archive candidate and y a complete rollout. A single hidden state is a feature representation, not an assertion that it contains the full Transformer attention cache.

ICWM learning and prediction residuals. Following the prediction-error approach to curiosity (Pathak et al., 2017), an encoder $\phi_\psi : \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}$ and predictor $f_\psi : \mathbb{R}^{2d_h} \to \mathbb{R}^{d_h}$ define

$$\hat{\phi}_{t+1} = f_\psi([\phi_\psi(s_t); a_t]), \qquad e_t(\psi) = \tfrac{1}{2}\|\phi_\psi(s_{t+1}) - \hat{\phi}_{t+1}\|_2. \tag{6}$$

Here $[\cdot;\cdot]$ denotes concatenation, and ψ collects the parameters of both networks. The encoder and predictor are jointly fitted as the ICWM on mini-batches $\mathcal{B} \subset \mathcal{D}_\tau$ using

$$\mathcal{L}_{\text{ICWM}}(\psi; \mathcal{B}) = \frac{\sum_{y \in \mathcal{B}} \sum_t m_t e_t(\psi)}{\sum_{y \in \mathcal{B}} \sum_t m_t}. \tag{7}$$

The norm in Equation (6) is the Euclidean norm, not its square. For mini-batches with no trainable tokens, set the cached residuals to zero and skip the ICWM update. Otherwise, for each mini-batch, we cache $e_t = \text{sg}[e_t(\psi)]$ before its optimizer step, then update ψ; sg denotes stop-gradient. Thus, the cached residuals are online prediction errors evaluated at successive ICWM parameter states, not a rescore of the epoch using one final predictor. No ICWM gradient is propagated into the actor.

The code-diversity diagnostic in Figure 1 compares generated code strings (Appendix F); it is not the ICWM residual e_t or bonus b_t, and does not enter the training objective. High ICWM prediction error identifies transitions that the current predictor models poorly. It does not establish semantic novelty, insufficient prior exploration, or future task improvement, and also depends on predictor capacity and representation drift. Moreover, joint training of the encoder and predictor with this loss alone admits constant-feature solutions. The formulation does not provide an anti-collapse guarantee; this limitation should be distinguished from the empirical task performance.

Epoch normalization. Let $\mathcal{I}_\tau = \{(y,t) : y \in \mathcal{D}_\tau, 1 \leqslant t \leqslant T_y\}$ be the set of actual completion positions, including inserted suffix positions but excluding prompts and padding. We define

$$\mu_\tau = \frac{1}{|\mathcal{I}_\tau|} \sum_{(y,t)\in\mathcal{I}_\tau} m_t e_t, \qquad \sigma_\tau^2 = \frac{1}{|\mathcal{I}_\tau|} \sum_{(y,t)\in\mathcal{I}_\tau} (m_t e_t - \mu_\tau)^2, \tag{8}$$

$$\hat{e}_t = \mathrm{sg}\left[\frac{m_t e_t - \mu_\tau}{\sqrt{\sigma_\tau^2 + \varepsilon}}\right], \qquad \varepsilon > 0. \tag{9}$$

Here μ_τ and σ_τ^2 are the epoch mean and variance of the masked residuals, and ε is a positive numerical stabilizer. This convention treats inserted tokens as zero residuals in the epoch statistics; it is not normalization over trainable tokens only. It centers the signal over $\mathcal{I}_\tau$ and rescales prediction errors, but does not ensure that their scale matches the task coefficients or that a subsequently masked signal remains centered.

Selective curiosity bonus. We retain the normalized residual only at sampled tokens outside the sampling policy's top-k choices:

$$u_t = m_t \, \mathbf{1}[y_t \notin \mathrm{TopK}_k(\pi_{\theta_{\mathrm{old}}}(\cdot \mid z_{<P+t-1}))], \tag{10}$$

The selected tokens receive the normalized prediction-error bonus

$$b_t = u_t \hat{e}_t. \tag{11}$$

Here $\mathbf{1}[\cdot]$ is the indicator function, and TopK_k returns the k highest-probability token identities. With $k = 1$, this selects departures from the highest-probability token, using the inference engine's tie-breaking convention. It is a heuristic for directing exploration, not an entropy threshold or a guarantee that selected positions are semantic branch points. The mask is recorded during generation and held fixed during training. In general, $\mathbb{E}[b_t] \neq 0$ even when $\mathbb{E}[\hat{e}_t] = 0$; we therefore call b_t a token-local bonus, not an unbiased intrinsic advantage. It is applied at the current token without accumulating future intrinsic rewards. Because normalization is centered, b_t can be positive or negative at selected tokens; unselected tokens still receive task feedback and can train the ICWM through m_t.

3.3 Joint Optimization and Training Procedure

Joint policy update. The update combines the task coefficient with the bonus in Equation (11):

$$\tilde{A}_t = \mathrm{sg}[A_t + \eta_\tau b_t]. \tag{12}$$

The curiosity weight follows

$$\eta_\tau = \eta_0 \max\left(\alpha, \tfrac{1}{2}[1 + \cos(\pi\tau/E)]\right). \tag{13}$$

Here E is the total number of training epochs, $\tau = 0, \ldots, E-1$ indexes epochs, $\eta_0 \geqslant 0$ is the initial weight, and $0 < \alpha \leqslant 1$ is the relative floor. The coefficient is annealed toward its floor over training. Starting from $\theta = \theta_{\mathrm{old}}$, one adapter optimizer step minimizes

$$\mathcal{L}_{\mathrm{PG}}(\theta) = -\sum_{y\in\mathcal{D}_\tau} \sum_{t=1}^{T_y} m_t \frac{\pi_\theta(y_t \mid z_{<P+t-1})}{\pi_{\theta_{\mathrm{old}}}(y_t \mid z_{<P+t-1})} \tilde{A}_t. \tag{14}$$

The denominator and $\tilde{A}_t$ are detached; only the numerator is differentiated. This is a token-level importance-weighted surrogate evaluated on the sampled batch. At $\theta = \theta_{\mathrm{old}}$, its gradient is the negative weighted token score-function sum. Finite-group weighting, group filtering, the sampled reference correction, and the token-local bonus distinguish this update from the exact gradient of the objective in Equation (1). Setting $\eta_0 = 0$ removes the curiosity bonus and recovers the task-only control update. Constant-reward groups are filtered before either loss, so they contribute no curiosity update either.

End-to-end training procedure. Algorithm 1 (Appendix B) summarizes each epoch. We collect and evaluate rollouts with θ_{old}, update the archive, and filter constant-reward groups. Retained rollouts receive task coefficients from Equations (2)–(4). When curiosity is enabled, we cache residuals before each ICWM update (Equations (6)–(7)), then normalize and mask them (Equations (8)–(11)). Finally, Equations (12)–(14) combine the signals for one adapter update; the actor remains fixed until this step.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Tasks and benchmarks. We evaluate on two task families. For **mathematical discovery**, following Yuksekgonul et al. (2026) and Novikov et al. (2025), we consider six open problems: Erdős minimum-overlap problem (**Erdős**, lower is better), three autocorrelation inequalities (**AC1**–**AC3**), the **Circle Packing** problem with n=26 circles in the unit square (sum of radii, higher is better; best-known $\approx$ 2.635983), and the **HadamardMatrix** task following Wang et al. (2025), with task settings specified in Appendix D. For **single-cell denoising**, we adopt the biology task of Yuksekgonul et al. (2026), in which the policy evolves a denoising program evaluated on two single-cell RNA sequencing (scRNA-seq) corpora, **PBMC** and **Tabula**, under three complementary criteria: a benchmark Score (higher is better), reconstruction MSE (lower is better), and Poisson negative log-likelihood (NLL; lower is better).

Baselines. We compare two baseline families: **frozen-LLM search**, comprising ALPHAEVOLVE (Novikov et al., 2025) via OPENEVOLVE (Sharma, 2025), SHINKAEVOLVE (Lange et al., 2026), BEST-OF-25600, and SIMPLETES (Ye et al., 2026); and **trainable test-time RL**, comprising THETAEVOLVE (Wang et al., 2025) and TTT-DISCOVER (Yuksekgonul et al., 2026). Frozen baselines vary candidates, prompts, and histories while keeping policy parameters fixed; SimpleTES uses independent trajectories, iterative refinement, local selection, and history reuse under 25,600 nominal generation slots. ThetaEvolve always denotes its test-time RL variant; both RL baselines update the policy from task feedback. TTT-Discover is the matched task-only control, using §3 with $\eta_0 = 0$, while `CURIO` adds ICWM-derived curiosity to test-time RL, comparing exploration among learning policies. We use Qwen3 backbones at 8B, 14B, and 235B (Yang et al., 2025); SimpleTES is evaluated at 8B and 14B, and the 235B comparison includes only TTT-Discover and `CURIO`. Budgets, optimization settings, and implementation details are provided in Appendix A.

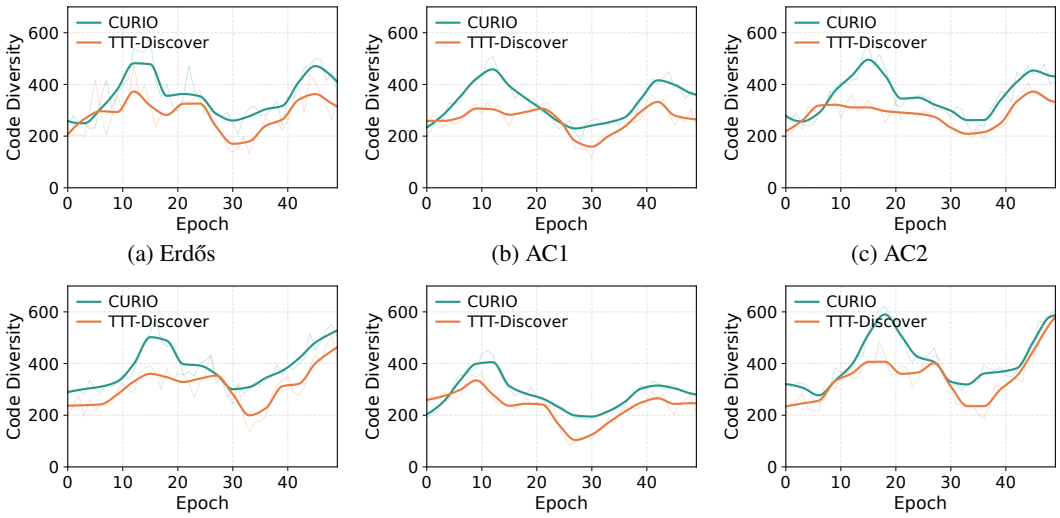
Evaluation protocol. For mathematical discovery, we report the best feasible raw objective found during each run, minimized for Erdős, AC1, and AC3, and maximized for AC2, Circle Packing, and Hadamard. For denoising, the epoch is selected by held-in MSE, and that epoch's Score, MSE, and Poisson NLL on PBMC and Tabula are reported together; neither held-out corpus is used for epoch selection (Appendix D.2). For every method, backbone, and task configuration in Tables 1 and 2, we report the arithmetic mean over three runs. Mathematical objectives are averaged after selecting the best feasible value within each run; denoising metrics are averaged after the within-run checkpoint selection described above. The main tables show means only for space; mean differences alone do not establish statistical significance. Prompts and evaluator specifications are in Appendix D and E.

4.2 TASK PERFORMANCE AND CODE DIVERSITY

Tables 1 and 2 summarize the reported results. **(i) Comparison with the task-only control.** `CURIO` improves on TTT-Discover for Erdős, AC1–AC3, and Hadamard at each tested scale, and matches the best reported performance on Circle Packing at the displayed precision. It also improves denoising Score and MSE on both held-out corpora. These are not uniform wins over every baseline: on 8B Tabula, ThetaEvolve has higher Score (0.7207 versus 0.7165) and lower MSE (0.1441 versus 0.1485).

(ii) Dependence on model scale. The Hadamard relative improvement over TTT-Discover is 4.11%, 3.36%, and 18.33% at 8B, 14B, and 235B, respectively. The largest observed advantage is therefore at 235B, but the trend is not monotonic. The absolute Erdős objective improves from 0.380932 to 0.380919 to 0.380880 as model size increases; this does not imply a growing relative advantage over the baseline. Attributing either trend specifically to richer hidden representations would require additional controls.

(iii) Code-level solution diversity. We measure code-level structural variation using OpenEvolve's *island code diversity metric* (Sharma, 2025), which compares pairs of generated programs through differences in code length, line count, and character sets. Appendix F defines the pairwise score and its aggregation. Figure 3 shows the measured trajectories across six mathematical discovery tasks. `CURIO` generally maintains higher code diversity D than TTT-Discover over the displayed epochs, although the curves cross or approach one another at some stages. Figure 1 summarizes the same



(a) Erdős (b) AC1 (c) AC2

(d) AC3 (e) Circle Packing (f) HadamardMatrix

Figure 3: **Code-level solution diversity across six discovery tasks.** Panels are ordered left to right, top to bottom: Erdős, AC1, AC2, AC3, Circle Packing, and HadamardMatrix. Each panel displays measured code diversity D over epochs for CURIO and TTT-Discover, using OpenEvolve's island code diversity metric (Sharma, 2025) (Appendix F); higher D indicates greater code-level structural variation.

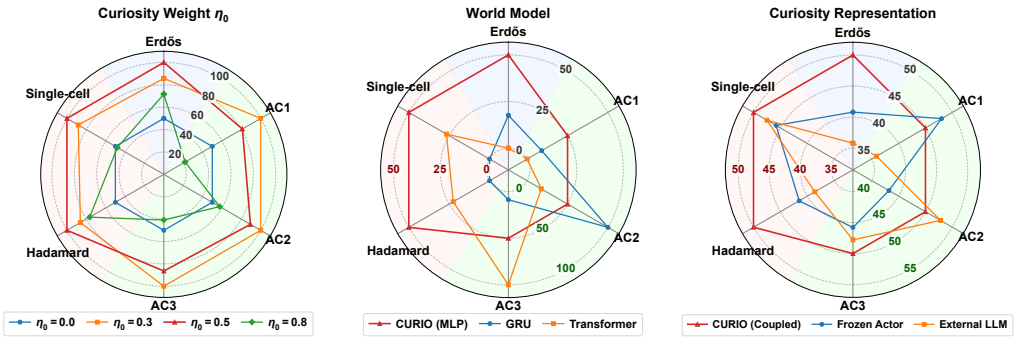
Table 1: **Test-time discovery across mathematical and combinatorial tasks.** Results are grouped by policy-backbone scale. All results are means over three runs; variability is omitted for space. Arrows denote improvement direction. Within each scale block, **best** is bold and second-best is underlined. Ties at the displayed precision share the same marking. CURIO improves over TTT-Discover on five objectives and matches the best reported performance on Circle Packing across model scales, with particularly strong gains on HadamardMatrix.

Scale	Method	Erdős (↓)	AC1 (↓)	AC2 (↑)	AC3 (↓)	Circle Packing (↑)	HadamardMatrix (↑)
8B	AlphaEvolve	0.381002	1.50561	0.8947	1.5062	2.63585	0.5589
	ThetaEvolve	0.380999	1.50649	0.9449	1.4942	2.63597	0.5748
	ShinkaEvolve	0.381026	1.51493	0.8501	1.5074	2.63597	0.5661
	Best-of-25600	0.381091	1.51032	0.9318	1.5081	2.63570	0.5554
	SimpleTES	0.381006	1.50628	0.9458	1.4951	2.63597	0.5776
	TTT-Discover	0.380995	1.50546	0.9463	1.4935	**2.63598**	0.5809
	CURIO	**0.380932**	**1.50513**	**0.9481**	**1.4846**	**2.63598**	**0.6048**
14B	AlphaEvolve	0.381068	1.50971	0.9158	1.5109	2.63482	0.5762
	ThetaEvolve	0.383710	1.51488	0.9351	1.6428	2.62491	0.5867
	ShinkaEvolve	0.381041	1.51052	0.9048	1.4996	2.63509	0.5826
	Best-of-25600	0.381119	1.50997	0.9271	1.5051	2.63439	0.5708
	SimpleTES	0.381009	1.50819	0.9374	1.4879	2.63546	0.5902
	TTT-Discover	0.380979	1.50608	0.9391	1.4592	**2.63598**	0.5864
	CURIO	**0.380919**	**1.50591**	**0.9418**	**1.4578**	**2.63598**	**0.6061**
235B	TTT-Discover	0.380918	1.50376	0.9562	1.4561	**2.63598**	0.7846
	CURIO	**0.380880**	**1.50315**	**0.9588**	**1.4549**	**2.63598**	**0.9284**

code-diversity diagnostic by training stage for Erdős and Circle Packing. Across the early (epochs 0–12), middle (13–25), late (26–38), and final (39–49) stages, CURIO has higher displayed D than TTT-Discover on both tasks. These observations support greater code-level structural diversity, but do not establish semantic diversity of the underlying constructions, unchanged token entropy, or the isolated contribution of non-top-k masking.

Table 2: **Single-cell denoising.** Results on PBMC and Tabula scRNA-seq corpora, grouped by policy-backbone scale. All results are means over three runs; variability is omitted for space. Arrows denote improvement direction. Within each scale block, **best** is bold and second-best is underlined. Ties at the displayed precision share the same marking. CURIO consistently improves over TTT-DISCOVER across model scales.

Scale	Method	PBMC			Tabula		
		Score (↑)	MSE (↓)	Poisson (↓)	Score (↑)	MSE (↓)	Poisson (↓)
8B	ALPHAEVOLVE	0.6738	0.1817	0.0554	0.6785	0.1716	0.0323
	THETAEVOLVE	0.6981	0.1588	0.0494	**0.7207**	**0.1441**	0.0302
	SHINKAEVOLVE	0.6787	0.1764	0.0535	0.6881	0.1669	0.0312
	BEST-OF-25600	0.6274	0.2071	0.0607	0.6382	0.1968	0.0354
	SIMPLETES	0.6958	0.1641	0.0503	0.7118	0.1514	0.0305
	TTT-DISCOVER	0.6842	0.2087	0.0818	0.6468	0.1963	0.0363
	CURIO	**0.7009**	**0.1552**	**0.0493**	0.7165	0.1485	**0.0301**
14B	ALPHAEVOLVE	0.6684	0.1821	0.0504	0.6586	0.1824	0.0302
	THETAEVOLVE	0.6814	0.1678	0.0503	0.6892	0.1586	0.0302
	SHINKAEVOLVE	0.6782	0.1715	0.0502	0.6783	0.1618	0.0301
	BEST-OF-25600	0.6171	0.2023	0.0506	0.6576	0.1822	0.0303
	SIMPLETES	0.6873	0.1651	0.0499	0.6966	0.1558	0.0302
	TTT-DISCOVER	0.6901	0.1615	0.0492	0.7012	0.1526	0.0301
	CURIO	**0.7052**	**0.1555**	**0.0491**	**0.7218**	**0.1423**	**0.0292**
235B	TTT-DISCOVER	0.7011	0.1562	0.0491	0.7254	0.1401	0.0291
	CURIO	**0.7156**	**0.1501**	**0.0490**	**0.7408**	**0.1334**	**0.0281**



(a) (b) (c)

Figure 4: **Ablation studies of CURIO on the Qwen3-14B backbone.** We report direction-aware normalized performance across six discovery tasks, with larger values indicating better performance. For each panel, the reference configuration is fixed at 50 on every task, so values above (below) 50 indicate improvement (degradation) relative to the reference. (a) *Curiosity weight*: we vary the initial curiosity weight $\eta_0 \in \{0, 0.3, 0.5, 0.8\}$, using $\eta_0 = 0$ (TTT-DISCOVER) as the reference. (b) *ICWM architecture*: we compare the ICWM's default multilayer perceptron (MLP) forward predictor with gated recurrent unit (GRU) and Transformer variants, using CURIO (MLP) as the reference. (c) *Curiosity representation*: we compare the ICWM's policy-coupled input representation with a frozen-actor representation and a frozen external-LLM representation, using CURIO (Coupled) as the reference. Single-cell averages the PBMC and Tabula results.

4.3 ABLATION STUDIES OF THE ICWM

To examine the sensitivity of the key design choices in CURIO, we conduct ablation studies on the Qwen3-14B backbone across six discovery tasks: Erdős, AC1, AC2, AC3, Hadamard, and single-cell denoising. We study three aspects of the ICWM: the bonus weight, the architecture of the ICWM's forward predictor, and the input representation used to model token-level transitions. Results are reported in Figure 4.

- **Curiosity Weight**: We vary the initial curiosity weight $\eta_0 \in \{0, 0.3, 0.5, 0.8\}$, where $\eta_0 = 0$ removes the curiosity reward and recovers TTT-DISCOVER. This evaluates the trade-off between task exploitation and curiosity-driven exploration.
- **ICWM Architecture**: We replace the ICWM's default MLP forward predictor with GRU- and Transformer-based predictors while keeping the remaining training procedure unchanged. This evaluates whether the gains depend on the predictor architecture within the ICWM.
- **Curiosity Representation**: We compare the ICWM's policy-coupled input representation with two decoupled alternatives: a frozen representation from the initial actor and a frozen representation from an independent external LLM. This evaluates whether the prediction-based curiosity signal benefits from modeling representations of the evolving policy itself.

Figure 4 indicates sensitivity to curiosity strength. Among the plotted settings, $\eta_0 = 0.3$ performs best on the autocorrelation tasks, while $\eta_0 = 0.5$ is preferable on Erdős, Hadamard, and the aggregated single-cell task. The largest tested weight, $\eta_0 = 0.8$, underperforms the task-only control on some tasks. Appendix A lists the task-specific weights selected from this sweep and the default for Circle Packing. The MLP reference is competitive with the recurrent and Transformer alternatives, but the best architecture depends on the task. Likewise, the coupled representation leads on most plotted tasks, with exceptions for frozen representations on individual tasks. These comparisons examine predictor and representation choices; they do not independently validate token-mask selection, rule out encoder collapse, or establish stability across random seeds.

5 Additional Related Work

LLM-based discovery combines program generation with automated evaluation and population search (Lehman et al., 2023; Meyerson et al., 2024; Romera-Paredes et al., 2024; Novikov et al., 2025; Sharma, 2025; Lange et al., 2026; Assumpção et al., 2025), with applications to reward design, equation discovery, GPU kernels, and agentic systems (Ma et al., 2024; Shojaee et al., 2025; Ouyang et al., 2025; Hu et al., 2025). Test-time RL additionally learns from experience on the current problem (Wang et al., 2025; Yuksekgonul et al., 2026); our ThetaEvolve and TTT-Discover baselines update their policies, whereas our SimpleTES baseline uses frozen-policy search with parallel trajectories, sequential refinement, local selection, and reusable histories (Ye et al., 2026). SimpleTES also studies trajectory-level post-training, which is distinct from that search baseline. Population search and policy learning are complementary. Exploration in RL (Sutton & Barto, 2018; Weng, 2020; Amin et al., 2021; Yu et al., 2024) ranges from undirected randomness and entropy regularization (Lillicrap et al., 2015; Dann et al., 2022; Andrychowicz et al., 2021; Yu et al., 2026; Cui et al., 2025; Shen, 2025) to directed bonuses based on visitation counts (Lai, 1987; Li et al., 2010; Jin et al., 2020) or prediction error (Pathak et al., 2017; Burda et al., 2018), with recent extensions to language models (Bai et al., 2025; Gao et al., 2025; Sun et al., 2025). In the best-candidate setting of test-time discovery, `CURIO` uses an ICWM to predict encoded actor-state transitions and applies its normalized residuals at non-top-k sampled tokens, complementing task-driven policy learning with a selective exploration signal.

6 Conclusion

We introduce `CURIO`, a test-time learning framework equipped with an ICWM. The ICWM learns token-level transitions in encoded policy hidden states, and its prediction residuals supply a curiosity bonus that supplements task feedback. The bonus is normalized within an epoch, retained at sampled tokens outside the policy's top-k choices, and combined with a token-local task coefficient under an annealed weight. The reported runs improve on the matched task-only control on several mathematical and denoising objectives, while matching the best reported performance on Circle Packing, and exhibit higher code-level structural diversity in the displayed analysis. Prediction error and code-level diversity remain proxies; further controls are needed to establish semantic exploration, the contribution of token selection, and robustness across runs.

AI USE DISCLOSURE

We used generative AI tools to assist with manuscript drafting and editing, LaTeX formatting, literature lookup, checks of mathematical exposition and consistency, and interpretation of reported results. The authors reviewed and edited the AI-assisted material and take full responsibility for the final content.

REFERENCES

Susan Amin, Maziar Gomrokchi, Harsh Satija, Herke Van Hoof, and Doina Precup. A survey of exploration methods in reinforcement learning. *arXiv preprint arXiv:2109.00157*, 2021.

Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2021.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Chenjia Bai, Yang Zhang, Shuang Qiu, Qiaosheng Zhang, Kang Xu, and Xuelong Li. Online preference alignment for language models via count-based exploration. *arXiv preprint arXiv:2501.12735*, 2025.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, et al. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*, 2025.

Chris Dann, Yishay Mansour, Mehryar Mohri, Ayush Sekhari, and Karthik Sridharan. Guarantees for epsilon-greedy reinforcement learning with function approximation. In *International conference on machine learning*, pp. 4666–4689. PMLR, 2022.

Jingtong Gao, Ling Pan, Yejing Wang, Rui Zhong, Chi Lu, Qingpeng Cai, Peng Jiang, and Xiangyu Zhao. Navigate the unknown: Enhancing llm reasoning with intrinsic motivation guided exploration. *arXiv preprint arXiv:2505.17621*, 2025.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Chi Jin, Zhuoran Yang, Zhaoran Wang, and Michael I Jordan. Provably efficient reinforcement learning with linear function approximation. In *Conference on learning theory*, pp. 2137–2143. PMLR, 2020.

Tze Leung Lai. Adaptive treatment allocation and the multi-armed bandit problem. *The annals of statistics*, pp. 1091–1114, 1987.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pp. 331–366. Springer, 2023.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pp. 661–670, 2010.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, et al. Eureka: Human-level reward design via coding large language models. In *International conference on learning Representations*, volume 2024, pp. 26516–26560, 2024.

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning*, 4(4):1–40, 2024.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787. PMLR, 2017.

Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. Posttrainbench: Can llm agents automate llm post-training? *arXiv preprint arXiv:2603.08640*, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

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

Han Shen. On entropy control in llm-rl algorithms. *arXiv preprint arXiv:2509.03493*, 2025.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan Reddy. Llm-sr: Scientific equation discovery via programming with large language models. In *International Conference on Learning Representations*, volume 2025, pp. 16054–16085, 2025.

Haoran Sun, Yekun Chai, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. Curiosity-driven reinforcement learning from human feedback. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23517–23534, 2025.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

Terence Tao. Mathematics in the age of ai. *arXiv preprint arXiv:2608.16753*, 2026.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Yuru Wang, Lejun Cheng, Yuxin Zuo, Sihang Zeng, Bingxiang He, Che Jiang, Junlin Yang, Yuchong Wang, Kaikai Zhao, Weifeng Huang, et al. Naturebench: Can coding agents match the published sota of nature-family papers? *arXiv preprint arXiv:2606.24530*, 2026.

Lilian Weng. Exploration strategies in deep reinforcement learning. *lilianweng.github.io*, Jun 2020. URL https://lilianweng.github.io/posts/2020-06-07-exploration-drl/.

Kai Wu, Hao Lyu, Zhen Luo, Chaofan Wang, Siyu Ye, Jinghao Lin, Xiaozhong Ji, Boyuan Jiang, Shengzhi Wang, Zihan Wang, et al. Ai4ai survey: From long-horizon agents to recursive self-improvement—definitions, reliable horizons, and open problems, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Rahul Thapa, Caiyin Yang, Chang Su, Rui Yang, Ruihua Liu, Rundao Li, Zeyu Li, Pengwei Sun, Chong Gao, Dachao Ding, Guangrong He, Miaolei Zhang, Lina Sun, Wenyang Wang, Yuchen Zhong, Zhuohao Shen, Puheng Li, Pan Lu, Bianxiao Cui, Di He, Jianzhu Ma, Junfeng Li, Hexi Baoyin, Yejin Choi, Stefano Ermon, Xiaowen Chu, Tongyang Li, Yuzhi Xu, and James Zou. Structured scaling of AI discovery across diverse scientific domains. *arXiv preprint arXiv:2604.19341*, 2026. doi: 10.48550/arXiv.2604.19341. URL https://arxiv.org/abs/2604.19341.

Fangxu Yu, Lai Jiang, Haoqiang Kang, Shibo Hao, and Lianhui Qin. Flow of reasoning: Training llms for divergent reasoning with minimal examples. *arXiv preprint arXiv:2406.05673*, 2024.

Fangxu Yu, Zinan Lin, Xiaodong Liu, Weijia Xu, Michael Xu, Tianyi Zhou, and Jianfeng Gao. Weak-to-strong on-policy distillation. *arXiv preprint arXiv:2607.26246*, 2026.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

APPENDIX CONTENTS

A Implementation Details **14**

B Training Procedure Details **16**

C Inference Procedure **16**

D Task Descriptions **18**

D.1 Mathematical discovery 18

D.2 Single-cell denoising 19

E Prompt Usage **20**

E.1 Shared state context 20

E.2 Mathematical discovery prompts 21

E.3 Single-cell denoising prompt 28

F Code-Level Solution Diversity Metric **30**

A Implementation Details

Experimental configuration. The configured discovery budget uses $E = 50$ epochs, a target of $N = 8$ parent groups per epoch, and $G = 64$ rollouts per group at temperature 1.0, corresponding to 25,600 generation slots. BEST-OF-25600 uses that nominal sample budget. Equal generation counts do not imply equal token counts, wall-clock time, or training cost. Archive bootstrap and realized counts are discussed below. We train low-rank adaptation (LoRA) adapters of rank 32 with Adam at learning rate 4×10^{-5} and reference-correction coefficient $\lambda = 0.1$, and the ICWM (encoder and forward predictor) with AdamW at 5×10^{-6} on mini-batches of 8 rollouts. The exploratory mask of Equation (10) uses $k = 1$ in every run, and the initial curiosity weight η_0 is the only task-dependent hyperparameter of CURIO: following the best settings in Figure 4(a), we use $\eta_0 = 0.5$ for Erdős, Hadamard, and denoising, and $\eta_0 = 0.3$ for AC1–AC3. Circle Packing retains the default $\eta_0 = 0.3$.

Table 3 summarizes the reported hyperparameters of CURIO. The method notation is defined in §3, and the training pseudocode is provided in Appendix B. The initial curiosity weight is task-dependent; the other listed settings are shared across tasks and backbone scales.

We expand the rows of Table 3 below. Each epoch of a run executes five stages in a fixed order (parent sampling, generation, evaluation, archive update, and training), and each stage checkpoints its output to disk before the next begins, so a run is resumable at epoch granularity after preemption.

Discovery loop. A run addresses one problem description d. The archive $\mathcal{H}$ is initialized with a seed candidate containing a reference program and its score: a published or hand-written construction for mathematical tasks, and MAGIC for denoising (§D.2). The configured budget is $E = 50$ epochs, a target of $N = 8$ parent groups per epoch, and $G = 64$ completions per group, giving $ENG = 25{,}600$ generation slots. This is a configured budget, not a consequence of the distinct-parent rule below: with only one seed, the archive cannot initially supply eight unrelated parents. An exact realized count requires a specified bootstrap or refill rule and verification against the run logs. Algorithm 1 therefore operates on the parents actually selected rather than assuming that the requested number is always available. Each request uses a sampling seed derived from its index within the epoch; different seeds do not guarantee different completions.

Archive and parent selection. Write v_p for the archive value of candidate p (the task objective, sign-flipped for minimization), n_p for its visit count, and q_p for the best child value observed so far, falling back to v_p if there is no scored child. Let V be the total number of rollouts drawn. The PUCT score is

$$\mathrm{PUCT}(p) = q_p + c\,\sigma_{\mathcal{H}}\,\rho_p \frac{\sqrt{1+V}}{1+n_p}, \qquad \rho_p = \frac{|\mathcal{H}| - \mathrm{rank}_0(v_p)}{\sum_{q\in\mathcal{H}}(|\mathcal{H}| - \mathrm{rank}_0(v_q))}. \tag{15}$$

Here $c = 1.0$, and rank_0 orders candidates from best to worst starting at zero, with ties resolved by a fixed ordering. The archive spread $\sigma_{\mathcal{H}}$ is the non-seed value range, floored at 10^{-6}; if fewer than two non-seed states exist, the floor is used. This makes the score well-defined for a singleton archive. These boundary conventions must be matched to the implementation when reproducing runs.

A rollout from p increments the visits of p and its ancestors. Parents are chosen greedily by PUCT, removing a selected candidate's ancestors and descendants from contention for that selection round. This rule can return fewer than N parents. Feasible, non-duplicate children are inserted, then the archive retains the two highest-valued children per parent and is truncated to its 1,000 highest-valued candidates, keeping the seed. Insertion does not require a child to improve on its parent.

Task coefficients and temperature calibration. Equation (2) defines the leave-one-out task coefficient in the main text. The finite-sample ratio is not generally unbiased. Groups with constant rewards are removed before both actor and ICWM training. For a retained group, temperature calibration targets

$$\mathrm{KL}(\mathrm{softmax}(\beta R) \,\|\, \mathrm{unif}_G) = \delta, \qquad \delta = \log 2,$$

using upper-bracket expansion and 60 bisection steps when a finite bracket exists. The target is not always attainable: with m tied maxima, the limiting divergence is $\log(G/m)$. A robust calibration routine must check this bound and use a documented finite cap or fallback instead of unbounded bracket expansion. The target and bisection count alone do not specify that boundary behavior.

Table 3: **Reported hyperparameters.** The discovery loop and policy optimization follow §2 and the task coefficients of §3.1; the ICWM-based curiosity block is specific to CURIO. Setting $\eta_0 = 0$ removes the intrinsic reward.

Hyperparameter	Value
Discovery loop	
Epochs E	50
Target parent groups per epoch N	8
Rollouts per parent G	64
Configured generation budget ENG	25,600
PUCT exploration constant c	1.0
Maximum archive size	1,000
Retained children per parent	2
Sampling temperature	1.0
Context window	32,768 tokens
Policy optimization	
LoRA rank / scaling factor / dropout	32 / 32 / 0
LoRA target modules	all attention and MLP projections, unembedding
Optimizer	Adam
Learning rate	4×10^{-5}
(β_1, β_2), ϵ	$(0.9, 0.95)$, 10^{-8}
Weight decay	0
Substeps per epoch	1
Reference-correction coefficient λ	0.1
Constant-reward groups	dropped
Optimizer state	persisted across epochs
ICWM and bonus	
Encoder ϕ_ψ / forward predictor f_ψ	three-layer MLP with rectified linear units (ReLU)
Forward-model intermediate width	backbone hidden size
Precision	`fp32`
Optimizer	AdamW
Learning rate	5×10^{-6}
(β_1, β_2)	$(0.9, 0.95)$
Mini-batch size	8 rollouts
Exploratory mask k	1
Annealing floor α	0.1
Initial curiosity weight η_0	0.5 (Erdős, Hadamard, denoising); 0.3 (others)
ICWM and optimizer state	checkpointed and reloaded each epoch
Infrastructure	
Generation backend	vLLM via Ray Data
Training backend	DeepSpeed
Evaluation	task-specific sandbox, isolated from training

The returned finite β is held fixed in the actor update. The token-local reference correction follows Equations (3)–(4), with the mean taken over trainable tokens in $\mathcal{D}_\tau$.

ICWM. The ICWM of §3.2 consists of the encoder and forward predictor. The encoder ϕ_ψ is a three-layer MLP $d_h \to 2d_h \to 2d_h \to d_h$ with ReLU activations, and the forward predictor f_ψ is a three-layer MLP $2d_h \to d_{\text{int}} \to d_{\text{int}} \to d_h$ that consumes the concatenation of the encoded state and the action embedding, with d_h the backbone hidden size and $d_{\text{int}} = d_h$ by default. Both run in `fp32` regardless of the backbone's compute dtype, since the residual e_t is a difference of nearby

vectors and is the quantity the intrinsic reward is read from. Within an epoch the ICWM is updated on mini-batches of 8 rollouts: each rollout's contribution to Equation (7) is backpropagated separately and normalized by the mini-batch's total trainable-token count, so gradients accumulate exactly as they would for a single batched loss while only one rollout's activation graph is resident at a time. The optimizer steps once per nonempty trainable mini-batch. The residuals used for policy training are cached and detached before each ICWM step. Equations (8)–(9) then normalize the cached, masked residuals once over $\mathcal{I}_\tau$: real completion positions of retained groups, including zero contributions from inserted suffixes and excluding prompts and padding. Residuals come from successive predictor states; they are not recomputed with a common final predictor. This normalization is epoch-wide rather than mini-batch-local. The ICWM weights and its optimizer state are written alongside the adapter at the end of every epoch and reloaded at the start of the next, so ICWM training continues across epochs rather than restarting. This persistence does not itself prevent feature collapse or guarantee decreasing prediction error.

Evaluation sandbox. Candidate programs are executed in a separate process, under a separate Python environment from the training stack, with no filesystem or network access and a per-task CPU allocation and wall-clock limit (Table 6). A candidate that raises, returns a malformed object, violates the task's feasibility constraints, or exceeds its wall-clock limit is assigned a correctness of zero and therefore a reward of zero; it is still recorded in the epoch's log but is never inserted into the archive. The evaluator's Python environment is pinned per task and is deliberately not upgraded in step with the training runtime, so that a runtime dependency bump cannot silently change what a score means. Each rollout's evaluation seed is its global index, which fixes the evaluator random seed independently of worker assignment. Wall-clock-limited candidate searches can still vary with hardware and scheduling.

B Training Procedure Details

Algorithm 1 gives the full training pseudocode corresponding to §3.3. At the start of an epoch, θ_{old} snapshots the actor. Generation, ICWM feature extraction, and the token-local reference correction all use that snapshot; no actor step occurs until the final policy update. Disabling the adapter yields the initial reference policy π_{θ_0}. The base weights remain frozen.

The ICWM mini-batch loop caches each residual before updating ψ, and the policy loss subsequently treats both the cached residuals and their epoch statistics as constants. Only the current-policy numerator in Equation (14) receives an actor gradient. Equation (7) trains the encoder and predictor but does not backpropagate through actor hidden states or action embeddings. Mini-batches with zero trainable tokens are skipped and have zero cached residuals. If the retained epoch has no trainable tokens, both training updates are skipped while the evaluated candidates and archive changes remain recorded.

Setting $\eta_0 = 0$ bypasses the curiosity block and sets $b_t = 0$, leaving the same task-only actor update. This equivalence concerns the matched TTT-Discover control; it does not assert that ThetaEvolve (test-time RL) shares the same training implementation. AlphaEvolve/OpenEvolve, ShinkaEvolve, Best-of-25600, and the SimpleTES search baseline keep their proposing policies fixed during discovery. Constant-reward filtering applies before the curiosity block, so the method does not provide intrinsic-only updates for those groups.

C Inference Procedure

Chat template. Every rollout is a single user turn. We do not inject a system prompt: the task specification and the state context are the entire user message, which keeps the prompt identical to what the evaluator's notion of the task implies and avoids a second place where behaviour could be steered. The assistant turn is opened with a `<think>` tag so that the backbone enters its reasoning mode deterministically rather than depending on the model to emit the tag itself. Generation stops at `<|im_end|>`. Table 4 gives the rendered form.

Table 4: **Chat template.** `{task_prompt}` is replaced by the per-task prompt of §E, which already contains the state context. No system message is rendered.

Algorithm 1 One epoch of CURIO: rollout collection and training.

Require: Archive $\mathcal{H}$; actor θ; reference θ_0; ICWM ψ; epoch τ
$\theta_{\text{old}} \leftarrow \theta$; compute η_τ by Equation (13)
Select available parents by the archive rule (Appendix A)
for each selected parent p **do**
  Render $x(d, p)$; sample G completions from $\pi_{\theta_{\text{old}}}$
  Record token log-probabilities, m_t, and u_t (Equation (10))
  Evaluate the extracted programs; record rewards and feasibility
end for
Insert feasible, non-duplicate candidates; prune the archive
Form $\mathcal{D}_\tau$ by dropping constant-reward groups
if $M_\tau > 0$ **then**
  Obtain each group's calibrated β; form A_i^{task} (Equation (2))
  Compute reference log-probabilities and A_t (Equations (3)–(4))
  if $\eta_0 > 0$ **then**
    Initialize cached residuals to zero; extract frozen h_j with a no-gradient forward of $\pi_{\theta_{\text{old}}}$
    for each mini-batch $\mathcal{B} \subset \mathcal{D}_\tau$ with $\sum_{y \in \mathcal{B}} \sum_t m_t > 0$ **do**
      Form transitions and cache detached, pre-update e_t (Equations (5), (6))
      Update ICWM parameters ψ using $\mathcal{L}_{\text{ICWM}}$ (Equation (7))
    end for
    Normalize cached residuals across $\mathcal{I}_\tau$ (Equations (8)–(9)); form b_t (Equation (11))
  else
    $b_t \leftarrow 0$ for every completion token
  end if
  Form detached $\tilde{A}_t$ (Equation (12))
  Take one adapter optimizer step on $\mathcal{L}_{\text{PG}}$ (Equation (14))
end if
Checkpoint the archive, actor, ICWM, and optimizer states

```
<|im_start|>user
{task_prompt}<|im_end|>
<|im_start|>assistant
<think>
```

Sampling. Rollouts are sampled at temperature 1.0 with no top-p or top-k truncation, within a 32,768-token context. Each request carries a seed equal to the rollout's index in the epoch. This fixes the sampling seed but does not ensure unique completions or bitwise reproducibility across execution backends. We request $k + 1$ candidate log-probabilities per position and retain the sampled token's log-probability. This suffices to identify membership in the top-k list under the engine's ordering; at $k = 1$ the requested candidate count is two. The exploratory mask u_t is computed from those returned log-probabilities during generation, so it is available to the trainer without any recomputation and without a second forward pass.

Two-phase budget forcing. A discovery rollout has to finish with an extractable program; a rollout that spends its entire context reasoning and never emits a code fence scores zero regardless of the quality of the reasoning. We therefore generate in two phases, applying budget forcing to the thinking span. Phase 1 samples with a cap on the thinking budget. If the rollout terminates normally, or has already emitted the answer marker "`‘python`, it is kept as is and no second phase runs. If instead phase 1 stopped because it hit its cap while still inside the think block, we append a forced suffix that closes the think block and opens the answer fence, and phase 2 resumes from the concatenated token sequence with whatever context remains, less a 50-token buffer. A rollout whose remaining budget is already exhausted at that point is recorded with finish reason `budget_exhausted` and evaluated on whatever it produced. Table 5 lists the two suffixes and the branch that selects them.

Two bookkeeping details matter for the rest of the method. The forced-suffix tokens were written by us rather than sampled, so they are excluded from the trainable mask ($m_t = 0$) and carry a placeholder log-probability; they therefore contribute no policy gradient. They are also assigned

$u_t = 0$, so they never attract an intrinsic reward. At sampled positions, the additional non-top-k condition determines whether the curiosity bonus is retained. Phase 2 is resumed from pre-tokenized ids rather than from re-tokenized text, so the boundary between the two phases cannot perturb the token sequence the trainer later scores.

Table 5: **Forced suffixes used by two-phase budget forcing.** The branch is decided by token subsequence matching on the phase-1 output, not by string matching, so it is robust to markers that do not round-trip through decoding.

phase-1 state	action
terminated normally (stop sequence emitted)	keep; no phase 2
"```python" already present in phase-1 output	keep; no phase 2
hit cap, still inside <think>	append "\n</think>\n\n```python\n"
hit cap, "</think>" is the final token	append "\n\n```python\n"
remaining context <= 0 after the suffix	record as budget_exhausted

Answer extraction. The evaluator takes the *last* fenced "‘python block in the completion as the candidate program. Taking the last rather than the first block is what makes the forced suffix well defined: a rollout that speculated about several programs while reasoning is scored on the one it committed to at the end. A completion with no fenced block is scored zero with the diagnostic "cannot extract python code from model response".

D Task Descriptions

Table 6 summarizes the seven discovery problems. All of them share the same interface: the policy emits a Python program, a sandbox executes it under a wall-clock limit and a CPU allocation, and a deterministic scorer returns a scalar. They differ in what the program must return, how the scalar is computed from it, and how that scalar is mapped to the reward the RL loop consumes.

Table 6: **Discovery tasks.** "Limit" is the hard wall-clock limit the sandbox enforces per candidate; the prompt separately advertises a soft compute budget to the model, and where the two differ the hard limit binds. "Reward" is the map from the raw objective r to the scalar the RL loop consumes; a candidate that fails the task's feasibility check receives reward 0 irrespective of r.

Task	Objective	Dir.	Limit	CPUs	Reward from raw score r
Erdős'	overlap constant C_5	↓	530 s	1	$1/(10^{-8}+r)$
AC1	1st autocorrelation bound	↓	1100 s	2	$1/(10^{-8}+r)$
AC2	2nd autocorrelation bound	↑	1100 s	2	r
AC3	3rd autocorrelation bound	↓	1100 s	2	$3\,\ell^3$, ℓ linear in $-r$
Circle Packing	sum of radii, n=26	↑	530 s	1	r
HadamardMatrix	$\lvert\det M\rvert/\det_{\text{ref}}$, n=29	↑	700 s	2	r
Denoising	reconstruction MSE	↓	600 s	2	$1/(10^{-8}+r)$

D.1 Mathematical discovery

Erdős' minimum-overlap problem. For a step function $h : [0,2] \to [0,1]$ normalized so that $\int_0^2 h = 1$, the objective is the overlap constant $C_5 = \max_k \int h(x)\,(1 - h(x+k))\,\mathrm{d}x$, which is to be minimized. The program returns a discretized h as n samples over $[0,2]$; the scorer computes $C_5 = \max\big(\mathrm{correlate}(h, 1-h)\big) \cdot \mathrm{d}x$ with $\mathrm{d}x = 2/n$ and rejects any candidate violating $0 \leqslant h_i \leqslant 1$ or not satisfying $\sum_i h_i = n/2$. The prompt states the best published bound $C_5 \leqslant 0.38092$ and sets 0.38080 as the target.

Autocorrelation inequalities. Three related problems ask for a sequence of step-function heights optimizing a ratio of norms of the autoconvolution. **AC1** minimizes $2n\,\|f * f\|_\infty / (\sum_i f_i)^2$ over

nonnegative sequences, target 1.5030. **AC2** maximizes $\|f * f\|_2^2/(\|f * f\|_1\|f * f\|_\infty)$, target 0.97. **AC3** minimizes $2n \max|f * f|/(\sum_i f_i)^2$ over sequences that *may take negative values*, target 1.4557; sign changes are the mechanism by which the autoconvolution peak is reduced, so unlike AC1 and AC2 the scorer does not clamp negatives. For all three, the exact scorer is given to the model inside the prompt (§E.2), so the policy optimizes against the function it will be graded by and the only uncertainty is the search. AC1 and AC2 additionally include a short prose summary of the published approach; AC3 includes the Host–Vinuesa window-times-oscillation template. AC3 uses a cubic shaping transform rather than a raw or reciprocal objective: its scores span a wide range early in a run, so the raw value is linearly mapped from $[1.4557, 3.2]$ onto $[1, 0]$, cubed, and scaled by 3, which concentrates reward differences near the target instead of near the initial construction.

Circle packing. Pack $n = 26$ circles into the unit square maximizing the sum of radii. The program returns centers and radii; the scorer runs a validator, given verbatim in the prompt, that rejects NaNs, negative radii, circles leaving $[0, 1]^2$, and any overlapping pair, each with a tolerance of 10^{-12}. The best-known value is ≈ 2.635983.

Hadamard matrix. Following the HadamardMatrix task of Wang et al. (2025), produce a 29×29 matrix with entries in $\{-1, +1\}$ maximizing $|\det M|$. The score is $|\det M|$ divided by the fixed benchmark normalization constant $\det_{\text{ref}} = 2^{28} \cdot 7^{12} \cdot 342$ for $n = 29$, matching their Appendix A.5. Here "Hadamard" is the benchmark label for a maximal-determinant search; an order-29 sign matrix is not a Hadamard matrix with mutually orthogonal rows. The quoted prompt suggests Paley and Sylvester constructions, local search over entry flips, and circulant structure as starting points.

D.2 Single-cell denoising

The policy evolves a Python function `magic_denoise(X, **kwargs)` that maps a raw scRNA-seq count matrix to denoised counts. We follow the OpenProblems dataset protocol used by Yuksekgonul et al. (2026): Pancreas supplies discovery feedback, while PBMC and Tabula Muris Senis Lung (Tabula) are held out. Within each dataset, observed molecules are split into input and target counts by binomial sampling. The reported Score is the mean of the benchmark-normalized MSE and Poisson scores, each normalized relative to no-denoising and perfect-denoising references. Two properties of this protocol matter for reading the results.

The training signal and the reported metric are different quantities. During the run, each candidate is scored on the held-in Pancreas corpus and its reconstruction MSE determines the reward and archive ranking. The numbers reported in §4 are instead the OpenProblems benchmark Score on two corpora, **PBMC** and **Tabula**, neither of which the loop ever observes. Transfer from the training corpus to the held-out ones is not monotone over a run, so per-epoch held-out curves are computed rather than read off the final epoch.

Held-out epochs are selected by the training objective, never by the held-out score. For each run we select the epoch by its in-domain MSE and report that epoch's PBMC and Tabula numbers together. Quoting each corpus at its own best epoch would describe a solution that does not exist and would select on the test set. Held-out evaluation uses `random_state=42`; repeating evaluation of a fixed candidate is distinct from repeating the stochastic discovery run.

Reward and the Poisson constraint. MSE is measured in log-normalized space after total-count normalization to 10^4; the reported Poisson metric is the NLL against the held-out molecules. On Pancreas, the normalized Poisson score acts as a hard gate rather than a reward term: a candidate is accepted only when $0.97 \leqslant (0.257575 - \text{poisson})/(0.257575 - 0.031739) \leqslant 1$, following Yuksekgonul et al. (2026), and the reward of an accepted candidate is $1/\text{MSE}$. The training gate is not applied to the held-out PBMC and Tabula results. On held-out data we clamp the normalized Poisson score to $[0, 1]$ following the OpenProblems definition and exclude candidates that produce a negative Poisson NLL, which would otherwise report a normalized score above 1 without being a valid denoiser. The seed program is the MAGIC baseline in its reversed-normalization form, which denoises in square-root space before normalizing; on the two held-out corpora it attains Score 0.6416 (PBMC) and 0.6414 (Tabula).

E PROMPT USAGE

Each rollout's prompt is assembled from two parts. A **static task specification** states the problem, the scorer, the return contract, and the resource limits; it is fixed for the lifetime of a run. A **dynamic state context** describes the particular archive node being expanded: the program that node holds, the score it achieved, how that compares with its own parent, the remaining gap to the target, and the tail of the program's standard output. The state context is what turns a fixed task description into a step of a search, by carrying previous candidates into the next prompt; policy and ICWM updates also carry experience across epochs. It is spliced into the middle of the specification, before the final instruction block.

§E.1 gives the state-context template, which is shared by all six mathematical tasks; §E.2 and §E.3 give the per-task specifications that wrap it. In every listing, `{braced}` names are substituted at render time. Long lines are wrapped for display, and a handful of non-ASCII glyphs in the original strings (`Erdős`, subscripted constants, arrows and inequality signs) are transliterated to ASCII here; the strings sent to the model are otherwise unchanged.

E.1 SHARED STATE CONTEXT

Table 7 gives the template. Three branches select the value report: a node with a parent gets the before/after pair, a node without one gets its own value, and the seed node, which has not been scored as a child of anything, gets only the target. Standard output is included because the tasks invite the model to instrument its own search; it is truncated to the last 500 characters (last 3,000 for denoising) with an explicit truncation marker, so a program that prints in a loop cannot crowd the specification out of the context. The trailing construction length is emitted for the autocorrelation tasks only, where sequence length is itself a search dimension.

Table 7: **State-context template shared by the six mathematical tasks.** `{metric}` and `{direction}` are per-task constants; the remaining fields come from the archive node being expanded. Exactly one of the three value branches is rendered.

````
You are iteratively optimizing {metric}.

  [if the node carries a program]
Here is the last code we ran:
```python
{parent_code}
```
  [else]
No previous code available.

  [branch 1: node has a parent and a value]
Here is the {metric} before and after running the code above
({direction} is better): {before:.6f} -> {after:.6f}
Target: {target}. Current gap: {gap:.6f}. Further improvements will also be
generously rewarded.

  [branch 2: node has a value but no parent]
Current {metric} ({direction} is better): {after:.6f}
Target: {target}. Current gap: {gap:.6f}. Further improvements will also be
generously rewarded.

  [branch 3: seed node]
Target {metric}: {target}

  [if the program printed anything]

--- Previous Program Output ---

                ...(TRUNCATED)...
{last 500 characters of stdout}
--- End Output ---

  [AC1, AC2, AC3 only]
Length of the construction: {n}
````

E.2 MATHEMATICAL DISCOVERY PROMPTS

Erdős' minimum-overlap problem. The specification states the continuous problem, the discretization the scorer assumes, and the two feasibility constraints, then names the published record and the target. The closing block is conditional: a node that already holds a program is told to do something structurally different from it, while the seed node is simply told to write code. Table 8 gives the prompt.

Table 8: **Erdős' minimum-overlap prompt.** `{budget_s}` is the advertised compute budget, `{n}` the length of the parent construction, and `{state_context}` the block of Table 7.

```
You are an expert in harmonic analysis, numerical optimization, and mathematical
discovery.
Your task is to find an improved upper bound for the Erdos minimum overlap
problem constant C_5.

## Problem

Find a step function h: [0, 2] -> [0, 1] that **minimizes** the overlap integral:

$$C_5 = \max_k \int h(x)(1 - h(x+k)) dx$$

**Constraints**:
1. h(x) in [0, 1] for all x
2. int_0^2 h(x) dx = 1

**Discretization**: Represent h as n_points samples over [0, 2].
With dx = 2.0 / n_points:
- 0 <= h[i] <= 1 for all i
- sum(h) * dx = 1 (equivalently: sum(h) == n_points / 2 exactly)

The evaluation computes: C_5 = max(np.correlate(h, 1-h, mode="full") * dx)

Smaller sequences with less than 1k samples are preferred - they are faster to
optimize and evaluate.

**Lower C_5 values are better** - they provide tighter upper bounds on the Erdos
constant.

## Budget & Resources
- **Time budget**: {budget_s}s for your code to run
- **CPUs**: 1 available

## Rules
- Define `run(seed=42, budget_s={budget_s}, **kwargs)` that returns
  `(h_values, c5_bound, n_points)`
- Use scipy, numpy, cvxpy[CBC,CVXOPT,GLOP,GLPK,GUROBI,MOSEK,PDLP,SCIP,XPRESS,
  ECOS], math
- Make all helper functions top level, no closures or lambdas
- No filesystem or network IO
- `evaluate_erdos_solution()` and `initial_h_values` (an initial construction,
  if available) are pre-imported
- Your function must complete within budget_s seconds and return the best
  solution found

**Lower is better**. Current record: C_5 <= 0.38092. Our goal is to find a
construction that shows C_5 <= 0.38080.

{state_context}

  [if a parent construction is available]
You may want to start your search from the current construction, which you can
access through the `initial_h_values` global variable (n={n} samples).
You are encouraged to explore solutions that use other starting points to
prevent getting stuck in a local optimum.

  [if the node carries a program]
Reason about how you could further improve this construction.
Ideally, try to do something different than the above algorithm. Could be using
different algorithmic ideas, adjusting your heuristics, adjusting / sweeping
your hyperparemeters, etc.
Unless you make a meaningful improvement, you will not be rewarded.

  [else]
```

```
Write code to optimize this construction.
```

First autocorrelation inequality. AC1 hands the model the scorer verbatim, so the objective is unambiguous, together with a prose restatement of the published linear-programming approach offered explicitly as inspiration rather than as a recipe. Table 9 gives the prompt.

Table 9: **AC1 prompt.** The embedded `evaluate_sequence` is the exact scorer used by the sandbox.

````
Act as an expert software developer and inequality specialist specializing in
creating step functions with certain properties.

Your task is to generate the sequence of non-negative heights of a step
function, that minimizes the following evaluation function:

```python
import numpy as np

def evaluate_sequence(sequence: list[float]) -> float:
    """
    Evaluates a sequence of coefficients with enhanced security checks.
    Returns np.inf if the input is invalid.
    """
    # --- Security Checks ---

    # Verify that the input is a list
    if not isinstance(sequence, list):
        return np.inf

    # Reject empty lists
    if not sequence:
        return np.inf

    # Check each element in the list for validity
    for x in sequence:
        # Reject boolean types (as they are a subclass of int) and
        # any other non-integer/non-float types (like strings or complex
        # numbers).
        if isinstance(x, bool) or not isinstance(x, (int, float)):
            return np.inf

        # Reject Not-a-Number (NaN) and infinity values.
        if np.isnan(x) or np.isinf(x):
            return np.inf

    # Convert all elements to float for consistency
    sequence = [float(x) for x in sequence]

    # Protect against negative numbers
    sequence = [max(0, x) for x in sequence]

    # Protect against numbers that are too large
    sequence = [min(1000.0, x) for x in sequence]

    n = len(sequence)
    b_sequence = np.convolve(sequence, sequence)
    max_b = max(b_sequence)
    sum_a = np.sum(sequence)

    # Protect against the case where the sum is too close to zero
    if sum_a < 0.01:
        return np.inf

    return float(2 * n * max_b / (sum_a**2))
```

A previous state of the art used the following approach. You can use it as
inspiration, but you are not required to use it, and you are encouraged to
explore.
```latex
Starting from a nonnegative step function $f=(a_0,\dots,a_{n-1})$ normalized so
that $\sum_j a_j=\sqrt{2n}$, set $M=\|f*f\|_\infty$. Next compute
$g_0=(b_0,\dots,b_{n-1})$ by solving a linear program, i.e.\ maximizing
$\sum_j b_j$ subject to $b_j\ge0$ and $\|f*g_0\|_\infty\le M$; as is standard,
````
````

```
the optimum is attained at an extreme point determined by an active set of
binding inequalities, here corresponding to important constraints where the
convolution bound $(f*g_0)(x)\le M$ is tight and limiting. Rescale $g_0$ to
match the normalization, $g=\frac{\sqrt{2n}}{\sum_j b_j}g_0$, and update
$f\leftarrow (1-t)f+t g$ for a small $t>0$. Repeating this step produces a
sequence with nonincreasing $\|f*f\|_\infty$, and the iteration is continued
until it stabilizes.
```

```
Your task is to write a search function that searches for the best sequence of
coefficients. Your function will have {budget_s} seconds to run, and after that
it has to have returned the best sequence it found. If after {budget_s} seconds
it has not returned anything, it will be terminated with negative infinity
points. All numbers in your sequence have to be positive or zero. Larger
sequences with {budget_s}s of items often have better attack surface, but too
large sequences with 100s of thousands of items may be too slow to search.

You may code up any search method you want, and you are allowed to call the
evaluate_sequence() function as many times as you want. You have access to it,
you don't need to code up the evaluate_sequence() function.

{state_context}

You may want to start your search from one of the constructions we have found so
far, which you can access through the 'height_sequence_1' global variable.
However, you are encouraged to explore solutions that use other starting points
to prevent getting stuck in a local minimum.

Reason about how you could further improve this construction.
Ideally, try to do something different than the above algorithm. Could be using
different algorithmic ideas, adjusting your heuristics, adjusting / sweeping
your hyperparemeters, etc.
Unless you make a meaningful improvement, you will not be rewarded.

Rules:
- You must define the `propose_candidate` function as this is what will be
  invoked.
- You can use scientific libraries like scipy, numpy, cvxpy[CBC,CVXOPT,GLOP,
  GLPK,GUROBI,MOSEK,PDLP,SCIP,XPRESS,ECOS], math.
- You can use up to 2 CPUs.
- Make all helper functions top level and have no closures from function
  nesting. Don't use any lambda functions.
- No filesystem or network IO.
- Do not import evaluate_sequence yourself. Assume it will already be imported
  and can be directly invoked.
- **Print statements**: Use `print()` to log progress, intermediate bounds,
  timing info, etc. Your output will be shown back to you.
- Include a short docstring at the top summarizing your algorithm.

Make sure to think and return the final program between ```python and ```.
```

Second autocorrelation inequality. AC2 maximizes rather than minimizes, and its scorer integrates the autoconvolution numerically, so the embedded function is longer than AC1's. Its literature note describes a coarse-to-fine scheme (stochastic global search, projected gradient ascent, then upsampling by entry repetition) rather than a linear program. Table 10 gives the prompt.

Table 10: **AC2 prompt.** Structure matches AC1; the scorer, the optimization direction, the literature note, and the required entry-point name differ.

```
Act as an expert software developer and inequality specialist specializing in
creating step functions with certain properties.

Your task is to generate the sequence of non-negative heights of a step
functions, that maximizes the following evaluation function:

```python

import numpy as np

def evaluate_sequence(sequence: list[float]) -> float:
    # Verify that the input is a list
    if not isinstance(sequence, list):
```
```

```
        raise ValueError("Invalid sequence type")

    # Reject empty lists
    if not sequence:
        raise ValueError("Empty sequence")

    # Check each element in the list for validity
    for x in sequence:
        # Reject boolean types (as they are a subclass of int) and
        # any other non-integer/non-float types (like strings or complex
        # numbers).
        if isinstance(x, bool) or not isinstance(x, (int, float)):
            raise ValueError("Invalid sequence element type")

        # Reject Not-a-Number (NaN) and infinity values.
        if np.isnan(x) or np.isinf(x):
            raise ValueError("Invalid sequence element value")

    # Convert all elements to float for consistency
    sequence = [float(x) for x in sequence]

    # Protect against negative numbers
    sequence = [max(0, x) for x in sequence]

    # Check if sum of sequence will be too close to zero
    if np.sum(sequence) < 0.01:
        raise ValueError("Sum of sequence is too close to zero.")

    # Protect against numbers that are too large
    sequence = [min(1000.0, x) for x in sequence]

    convolution_2 = np.convolve(sequence, sequence)
    # --- Security Checks ---

    # Calculate the 2-norm squared: ||f*f||_2^2
    num_points = len(convolution_2)
    x_points = np.linspace(-0.5, 0.5, num_points + 2)
    x_intervals = np.diff(x_points) # Width of each interval
    y_points = np.concatenate(([0], convolution_2, [0]))
    l2_norm_squared = 0.0
    for i in range(len(convolution_2) + 1):  # Iterate through intervals
        y1 = y_points[i]
        y2 = y_points[i+1]
        h = x_intervals[i]
        # Integral of (mx + c)^2 = h/3 * (y1^2 + y1*y2 + y2^2) where
        # m = (y2-y1)/h, c = y1 - m*x1, interval is [x1, x2], y1 = mx1+c,
        # y2=mx2+c
        interval_l2_squared = (h / 3) * (y1**2 + y1 * y2 + y2**2)
        l2_norm_squared += interval_l2_squared

    # Calculate the 1-norm: ||f*f||_1
    norm_1 = np.sum(np.abs(convolution_2)) / (len(convolution_2) + 1)

    # Calculate the infinity-norm: ||f*f||_inf
    norm_inf = np.max(np.abs(convolution_2))
    C_lower_bound = l2_norm_squared / (norm_1 * norm_inf)
    return C_lower_bound
```

A previous state of the art used the following approach. You can use it as inspiration, but you are not required to use it, and you are encoraged to explore.

```latex
Their procedure is a coarse-to-fine optimization of the score. It starts with a stochastic global search that repeatedly perturbs the current best candidate and keeps the perturbation whenever it improves (Q), with the perturbation scale gradually reduced over time. Once a good basin is found, they switch to a deterministic local improvement step, performing projected gradient ascent (move in the gradient direction and project back to the feasible region). To reach higher resolution, they lift a good low-resolution solution to a higher-dimensional one by simply repeating its entries and then rerun the local refinement. Iterating this explore-refine-upscale cycle yields their final high-resolution maximizer and the improved lower bound.
```

Your task is to write a search function, construct_function(), that searches for the best sequence of coefficients. Your function will have {budget_s} seconds to run, and after that it has to have returned the best sequence it found. If after {budget_s} seconds it has not returned anything, it will be terminated with negative infinity points. All numbers in your sequence have to be positive or

```
zero. Larger sequences with {budget_s}s of items often have better attack
surface, but too large sequences with 100s of thousands of items may be too slow
to search.

You may code up any search method you want, and you are allowed to call the
evaluate_sequence() function as many times as you want. You have access to it,
you don't need to code up the evaluate_sequence() function.

{state_context}

You may want to start your search from one of the constructions we have found so
far, which you can access through the 'height_sequence_1' global variable.
However, you are encouraged to explore solutions that use other starting points
to prevent getting stuck in a local minimum.

Reason about how you could further improve this construction.
Ideally, try to do something different than the above algorithm. Could be using
different algorithmic ideas, adjusting your heuristics, adjusting / sweeping
your hyperparemeters, etc.
Unless you make a meaningful improvement, you will not be rewarded, if you are
stuck you should think about how to get unstuck.

Rules:
- You must define the `construct_function` function as this is what will be
  invoked.
- You can use scientific libraries like scipy, numpy, cvxpy[CBC,CVXOPT,GLOP,
  GLPK,GUROBI,MOSEK,PDLP,SCIP,XPRESS,ECOS], math.
- You can use up to 2 CPUs.
- Make all helper functions top level and have no closures from function
  nesting. Don't use any lambda functions.
- No filesystem or network IO.
- Do not import evaluate_sequence yourself. Assume it will already be imported
  and can be directly invoked. Do not import height_sequence_1 yourself; it will
  already be available.
- **Print statements**: Use `print()` to log progress, intermediate bounds,
  timing info, etc. Your output will be shown back to you.
- Include a short docstring at the top summarizing your algorithm.

Make sure to think and return the final program between ```python and ```.
```

Third autocorrelation inequality. AC3 is the only task whose feasible set includes negative values, and the prompt says so three times (in the scorer's docstring, in the literature note, and in the construction guidelines) because clamping to nonnegative values is the natural habit carried over from AC1 and AC2 and it removes the mechanism that makes the problem tractable. Table 11 gives the prompt.

Table 11: **AC3 prompt.** `{target}` is 1.4557.

````
You are an expert in computational optimization and harmonic analysis. Your task
is to design a Python program that constructs a discrete function `f` on the
domain [-1/4, 1/4] to minimize the third autocorrelation constant C3, aiming to
beat the SOTA of {target}.

C3 is computed by the following evaluation function (lower is better):

```python
import numpy as np

def evaluate_sequence(sequence: list[float]) -> float:
    """
    Evaluates a discrete function f for the THIRD autocorrelation inequality.
    Computes C3 = abs(2 * n * max(|conv(f, f)|) / (sum(f))**2), lower is better.

    NOTE: function values MAY be negative. Sign changes (phase cancellation)
    are a key mechanism for reducing the autoconvolution peak. Only NaN/inf,
    extreme magnitudes (|x| > 1e10), and a near-zero sum are rejected.
    """
    arr = np.asarray([float(x) for x in sequence], dtype=float)
    if not np.all(np.isfinite(arr)):
        return np.inf
    if np.max(np.abs(arr)) > 1e10:
        return np.inf
````
````

```
    total = float(np.sum(arr))
    if abs(total) < 1e-10:
        return np.inf
    conv = np.convolve(arr, arr)
    max_conv_abs = float(np.max(np.abs(conv)))
    return float(abs(2 * len(arr) * max_conv_abs / (total ** 2)))
```

Key Insight from Mathematical Literature (Host, Vinuesa):
The best-known constructions are based on the product of a smooth, oscillating function and a window function with compact support. A highly successful continuous analog is `f(x) = (1 + cos(2*pi*x))` for `x` in `[-1/4, 1/4]` and `0` otherwise. IMPORTANT: function values MAY be negative - sign changes (phase cancellation) are a key mechanism for reducing the autoconvolution peak. Do NOT clamp your function to be non-negative.

Construction Guidelines:
1. **Window Function:** Define a "window"/"support" for your function, centered and occupying a fraction of the total domain (e.g. the middle 50%, like `n//4` to `3*n//4`). The function is zero outside this window.
2. **Oscillatory Component:** Inside the window use a smooth, symmetric, oscillating pattern. `A * (1 + B * cos(C * x))` is a powerful template (with B>1 it takes negative values).
3. **Parameterization:** Explore `support_width`, amplitude `A`, modulation `B`, frequency `C`.
4. **Discretization:** Map the continuous form onto a discrete grid (a length around 400-800 works well; n~600 is a good default). Mind boundary conditions at the window edges.

Refinement strategy (to push past SOTA):
- Perform many local-search iterations (e.g. 5,000-20,000) on the current candidate.
- Multi-stage adaptive step size: start with larger changes (+/-0.05), then shrink (+/-0.01, +/-0.001) to fine-tune.
- Targeted search: find the indices where `conv(f, f)` has the highest absolute values and focus perturbations there - they dominate C3.
- Escape local minima: use a simulated-annealing-style schedule; if stagnant for hundreds of iterations, apply a larger random perturbation.
- Sign flipping: systematically test flipping the signs of small segments - phase cancellation is the key mechanism for lowering the convolution peak.

Your function will have {budget_s} seconds to run and must return the best sequence (a Python list of floats, negatives allowed) it found. You may call evaluate_sequence() as many times as you want - it is already available, do not redefine or import it.

{state_context}

You may start your search from one of the constructions found so far via the `height_sequence_1` global variable, but you are encouraged to explore other starting points to avoid local minima.

Reason about how to further improve the construction. Try something different from the previous algorithm (different structure, heuristics, or hyperparameters). Unless you make a meaningful improvement, you will not be rewarded.

Rules:
- You must define the `propose_candidate` function as this is what will be invoked.
- You can use scientific libraries like scipy, numpy, cvxpy[CBC,CVXOPT,GLOP, GLPK,GUROBI,MOSEK,PDLP,SCIP,XPRESS,ECOS], math.
- You can use up to 2 CPUs.
- Make all helper functions top level and have no closures from function nesting. Don't use any lambda functions.
- No filesystem or network IO.
- Do not import evaluate_sequence yourself. Assume it will already be imported and can be directly invoked.
- **Print statements**: Use `print()` to log progress, intermediate bounds, timing info, etc. Your output will be shown back to you.
- Include a short docstring at the top summarizing your algorithm.

Make sure to think and return the final program between ```python and ```.

Circle packing. The circle-packing prompt is the shortest of the six because the problem needs no restatement; what it does include is the validator's source, extracted at render time from the evaluator module so that the text in the prompt cannot drift from the code that runs. This is also the one task

whose closing instruction asks for an explicit plan, in <strategy> tags, before the program. Table 12 gives the prompt.

Table 12: **Circle-packing prompt.** {n} is the number of circles (26 in all reported runs). The validator is inlined from the evaluator's source at render time.

````
You are an expert mathematician specializing in circle packing problems and
computational geometry.

Your task is to pack {n} circles in a unit square [0,1]x[0,1] to maximize the
sum of radii.

We will run the below validation function (read-only, do not modify this):
```python
def validate_packing(centers, radii):
    """
    Validate that circles don't overlap and are inside the unit square

    Args:
        centers: np.array of shape (n, 2) with (x, y) coordinates
        radii: np.array of shape (n) with radius of each circle

    Returns:
        True if valid, False otherwise
    """
    n = centers.shape[0]

    # Check for NaN values
    if np.isnan(centers).any():
        print("NaN values detected in circle centers")
        return False

    if np.isnan(radii).any():
        print("NaN values detected in circle radii")
        return False

    # Check if radii are nonnegative and not nan
    for i in range(n):
        if radii[i] < 0:
            print(f"Circle {i} has negative radius {radii[i]}")
            return False
        elif np.isnan(radii[i]):
            print(f"Circle {i} has nan radius")
            return False

    # Check if circles are inside the unit square
    for i in range(n):
        x, y = centers[i]
        r = radii[i]
        if x - r < -1e-12 or x + r > 1 + 1e-12 or y - r < -1e-12 \
                or y + r > 1 + 1e-12:
            print(f"Circle {i} at ({x}, {y}) with radius {r} is outside the "
                  f"unit square")
            return False

    # Check for overlaps
    for i in range(n):
        for j in range(i + 1, n):
            dist = np.sqrt(np.sum((centers[i] - centers[j]) ** 2))
            if dist < radii[i] + radii[j] - 1e-12:  # Allow for tiny errors
                print(f"Circles {i} and {j} overlap: dist={dist}, "
                      f"r1+r2={radii[i]+radii[j]}")
                return False

    return True
```

{state_context}

Reason about how you could further improve this packing. Consider:
- Are circles placed optimally near boundaries and corners?
- Could a different arrangement (hexagonal, nested, hybrid) yield better
  results?
- Are there gaps that could be filled with repositioned or resized circles?
- Could optimization parameters or methods be improved?
````

```
Rules:
- You must define the run_packing function:
  def run_packing() -> tuple[np.ndarray, np.ndarray, float]
- Returns (centers, radii, sum_radii) where centers has shape ({n}, 2) and radii
  has shape ({n},).
- You can use scientific libraries like scipy, numpy, cvxpy, math.
- Centers must lie within [0,1]^2 and radii must be nonnegative.
- The pair (centers, radii) must satisfy non-overlap and boundary constraints.
- Make all helper functions top level and have no closures from function
  nesting. Don't use any lambda functions.
- No filesystem or network IO.
- You need to get really creative and think from first principles.

Make sure to /think step by step, first give your strategy between <strategy>
and </strategy> tags, then finally return the final program between ```python
and ```.
```

Hadamard matrix. The Hadamard prompt states the normalization explicitly (the reported score is $|\det M|$ over the fixed benchmark normalization constant for $n = 29$, not the determinant itself) and lists construction families rather than an algorithm. Table 13 reproduces the original prompt, including its informal use of "Hadamard matrix" and "theoretical maximum"; these phrases do not establish an attainable optimum.

Table 13: **Hadamard matrix prompt.**

```
You are an expert mathematician specializing in combinatorial optimization and
Hadamard matrices.

A Hadamard matrix is an n x n matrix with entries +1 or -1 such that the
absolute value of the determinant is maximized. For n=29, the theoretical
maximum determinant is 127069834656817034035 2 (this is (2^28) * (7^12) * 342).

Your task is to write a Python function `generate_hadamard_matrix(n)` that
returns a 29x29 matrix with entries +1 or -1 that maximizes the absolute
determinant. The score is |det(M)| / theoretical_max, where higher is better.

Key approaches to consider:
- Start from known Hadamard matrix constructions (Paley, Sylvester, etc.)
- Use local search: flip entries and check if determinant improves
- Use simulated annealing or genetic algorithms
- Exploit mathematical structure (circulant matrices, quadratic residues)
- For n=29, Paley construction using quadratic residues mod 29 is a strong
  starting point

Your function will have 600 seconds to run. The function should return a numpy
array of shape (29, 29) with entries +1 or -1.

{state_context}

Rules:
- You must define the `generate_hadamard_matrix(n)` function as this is what
  will be invoked.
- You can use scientific libraries like scipy, numpy, math.
- You can use up to 2 CPUs.
- Make all helper functions top level and have no closures from function
  nesting. Don't use any lambda functions.
- No filesystem or network IO.
- **Print statements**: Use `print()` to log progress, intermediate bounds,
  timing info, etc. Your output will be shown back to you.
- Include a short docstring at the top summarizing your algorithm.

Make sure to think and return the final program between ```python and ```.
```

E.3 SINGLE-CELL DENOISING PROMPT

The denoising prompt includes domain-specific guidance beyond the problem statement. The closing "Key Insights" block records four empirical facts about the benchmark, the most important being that denoising must happen in raw or square-root space *before* normalization, which is the difference

between a normalized Poisson deviance of ≈ 0.98 and ≈ 0.55 and therefore between passing and failing the hard gate. Without it, essentially every early candidate is rejected on the constraint and the run receives no gradient at all. The prompt also differs from the mathematical tasks in that its state context reports two metrics rather than one, and it retains a longer tail of standard output (3,000 characters), since candidates here routinely print per-stage diagnostics. Table 14 gives the prompt.

Table 14: **Single-cell denoising prompt.** The two evaluation functions are the exact scorers. `{mse}`, `{poisson}`, `{norm}` and `{gap}` are the parent node's measurements; the seed node reports the MAGIC baseline at MSE 0.2316.

```
You are an expert in computational biology and single-cell RNA-seq analysis.
Your task is to develop a denoising algorithm for scRNA-seq count data.

## Problem

Single-cell RNA-seq data is noisy due to technical dropout and low capture
efficiency.
Given noisy count data, predict the true expression levels.

Your prediction is evaluated against held-out molecules using two metrics:
1. **MSE** - Mean Squared Error in log-normalized space (lower is better, this
   is the reward)
2. **Poisson Loss** - Poisson negative log-likelihood (hard constraint)

## Data Format

- Input `X`: numpy array of shape (n_cells, n_genes) - raw count data
- Output: numpy array of same shape - your denoised counts

## Evaluation

Your output is evaluated using these exact functions:

```python
def evaluate_mse(test_data, denoised):
    test_X = scprep.utils.toarray(test_data).copy()
    denoised_X = np.asarray(denoised).copy()
    test_adata = anndata.AnnData(X=test_X)
    denoised_adata = anndata.AnnData(X=denoised_X)
    sc.pp.normalize_total(test_adata, target_sum=10000)
    sc.pp.log1p(test_adata)
    sc.pp.normalize_total(denoised_adata, target_sum=10000)
    sc.pp.log1p(denoised_adata)
    return sklearn.metrics.mean_squared_error(test_adata.X, denoised_adata.X)
```

```python
def evaluate_poisson(train_data, test_data, denoised):
    test_X = scprep.utils.toarray(test_data)
    denoised_X = np.asarray(denoised).copy()
    initial_sum = train_data.sum()
    target_sum = test_X.sum()
    denoised_scaled = denoised_X * target_sum / initial_sum
    return poisson_nll_loss(test_X, denoised_scaled)
```

Scoring

Poisson is a HARD CONSTRAINT. Your solution is REJECTED if
`poisson_norm < 0.97`.
- `poisson_norm = (0.257575 - poisson) / (0.257575 - 0.031739)`
- MAGIC baseline achieves poisson_norm ~ 0.97

Reward = 1/MSE (after passing Poisson constraint). Lower MSE = higher
reward.

Budget & Resources

- **Time budget**: 400s for your code to run
- **CPUs**: 2 available

Function Signature

```python
```
```

```
def magic_denoise(X, **kwargs):
    # kwargs may include: budget_s, random_state, knn, t, n_pca, solver, decay,
    #                     knn_max, n_jobs
    # Your implementation here
    return denoised_X  # same shape as X
```

Rules

- Implement `magic_denoise(X, ...)` that returns a denoised array of the same
  shape
- Available libraries: numpy, scipy, sklearn, graphtools, scprep, scanpy,
  anndata
- Make all helper functions top-level (no closures or lambdas)
- No filesystem or network IO

Key Insights from Benchmarks

- NORMALIZATION ORDER MATTERS: Denoise in raw/sqrt space first, then normalize.
  "Reversed normalization order" achieves Poisson ~0.98 vs ~0.55 for standard
  order.
- Square root transform is variance-stabilizing for Poisson distributions
- Poisson loss is highly affected by low non-zero values - push values < 1
  toward zero
- The original MAGIC with reversed normalization achieves Poisson ~ 0.97 (the
  constraint threshold)

You are iteratively improving a scRNA-seq denoising algorithm (minimize MSE).

Here is the current implementation:
```python
{parent_code}
```

Current metrics (lower is better): MSE={mse:.6f}, Poisson={poisson:.6f}
(norm={norm:.4f})
Target MSE: 0.2. Current gap: {gap:.6f}. Further improvements will also be
generously rewarded.

  [if the program printed anything]
Output from last run:
```
{last 3000 characters of stdout}
```

Write your improved `magic_denoise` function.
```

---

# F CODE-LEVEL SOLUTION DIVERSITY METRIC

**Pairwise code comparison.** Figure 3 uses OpenEvolve's *island code diversity metric* (Sharma, 2025). For code strings $c_i, c_j$, its structural dissimilarity is

$$d_{\text{code}}(c_i, c_j) = 0.1\,\big|L(c_i) - L(c_j)\big| + 10\,\big|N(c_i) - N(c_j)\big| + 0.5\,\big|S(c_i) \triangle S(c_j)\big|. \tag{16}$$

Here $L(c)$ is the number of characters in the code string, $N(c)$ counts newline characters, and $S(c)$ is the set of distinct characters. The symmetric difference $S(c_i) \triangle S(c_j)$ contains character types appearing in exactly one string. OpenEvolve implements the line-count term with `code.count("\n")`, rather than counting nonempty lines.

**Aggregation.** Let $\mathcal{Q}_\tau$ denote the selected unordered program pairs at epoch $\tau$. The island statistic is their mean dissimilarity:

$$D_\tau = \frac{1}{|\mathcal{Q}_\tau|} \sum_{(i,j)\in\mathcal{Q}_\tau} d_{\text{code}}(c_i, c_j), \tag{17}$$

with $D_\tau = 0$ when no pair is available. OpenEvolve's island routine sorts programs by identifier, considers up to five programs, and averages up to six pairwise comparisons. Its separate candidate-
```

level routine averages $d_{\text{code}}(c, r)$ over reference programs $r \in \mathcal{R}$ whose code differs from c, returning zero for an empty comparison set. These are different aggregations of the same pairwise heuristic.[1]

Interpretation. The score is nonnegative and unnormalized; larger values indicate greater differences in these code features. It is not an execution-based or semantic distance: formatting and comments can change it without changing behavior, while different programs can share the same length, newline count, and character set and receive zero dissimilarity. Accordingly, we interpret Figure 3 as code-level structural variation and compare methods within each task. Figure 1 reports stage-level summaries of this code-diversity diagnostic on Erdős and Circle Packing: early (epochs 0–12), middle (13–25), late (26–38), and final (39–49). Larger displayed values indicate greater structural variation, consistent with the epoch-level trajectories in Figure 3.

[1] Source: https://github.com/algorithmicsuperintelligence/openevolve/blob/main/openevolve/database.py, routines `_fast_code_diversity`, `_calculate_island_diversity`, and `_get_cached_diversity`.