DISCOVERING PRACTICAL ALGORITHMS FOR LEAST SQUARES USING LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Least squares is a central problem in scientific computing with applications across many domains such as the natural and social sciences. The design and implementation of state-of-the-art algorithms for solving least squares are often based on worst-case theoretical analysis and tailored to provide numerically stable solutions. These principles are often too conservative in real-world settings, incurring severe runtime slowdowns. Manually designing per-use-case algorithm implementations, on the other hand, requires domain knowledge and tedious trial-and-error. In this work, we leverage recent progress in large language model-guided automated discovery to automate the design of practical algorithms for numerical linear algebra applications. More specifically, our methodology frames the search for practical least squares algorithms as a meta-discovery problem: given a family of downstream solver tasks, we combine language model-guided algorithm evolution with robust meta-evaluation accounting for runtime and solver accuracy. We demonstrate the effectiveness and flexibility of our framework by discovering faster practical algorithms across an extensive list of synthetic and real-world least squares tasks. The discovered algorithms are $2\times$ faster than optimized baselines on large kernel regression tasks from the SUSY dataset for high-energy particle physics classification with up to one million rows. Additionally, we analyze the algorithms discovered by our framework and decompose the runtime improvements across algorithmic changes, hyperparameter optimization, and code engineering.

1 INTRODUCTION

Data-driven fitting of a linear relationship by minimizing the squared errors is one of the most studied problems in scientific computing, with wide-ranging applications including machine learning (Hastie et al., 2009; Rudi et al., 2017), economics (Wooldridge, 2009), physics (Bevington & Robinson, 2003; Aster et al., 2005), medical sciences (Hansen et al., 2021; Vittinghoff et al., 2005), astronomy (Feigelson & Babu, 2012) and many more. This is known as the *least squares problem* (LS), where given a matrix $A \in \mathbb{R}^{m\times n}$ with $m > n$ and a vector $b \in \mathbb{R}^m$, the goal is to find

$$x^* = \arg\min_{x\in\mathbb{R}^n} \|Ax - b\|_2^2. \tag{1}$$

Many algorithms with strong theoretical guarantees have been developed to solve this problem, such as backward-stable direct solvers based on QR or SVD factorization (Golub, 1965), faster but less accurate direct solvers based on the Cholesky factorization of the Gram matrix (Golub & Van Loan, 2013), and randomized preconditioned iterative algorithms based on sketching (Murray et al., 2023). They have highly optimized software implementations in C/Fortran that are directly wrapped and made available in popular programming languages (Dongarra et al., 1990; Anderson et al., 1999).

Despite the maturity of this field from both a mathematical and software perspective, the gap between theory and practice is wide when it comes to runtime speed (Figure 1, right). Many algorithms and implementations are developed with a worst-case perspective, where the priority is to find numerically stable high-accuracy solutions on all inputs. This leads to conservative choices that are often excessively slow for practical applications. The worst-case viewpoint is ill-suited to bridge this gap: the space of feasible LS solvers is large and includes a variety of algorithmic families with different trade-offs. Overly conservative worst-case settings often neglect better algorithmic designs, which outperform on (still broad) application domains.

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.

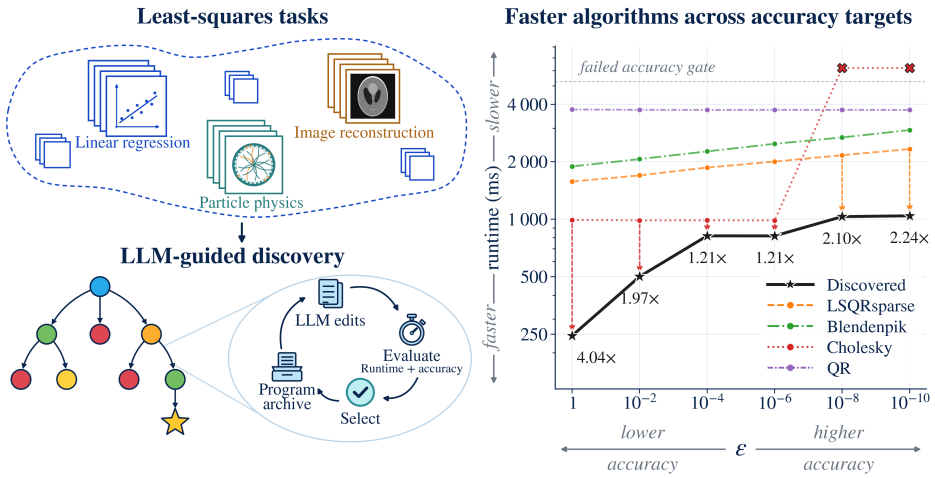

Figure 1: **Left.** Given a specific family of least squares problem classes, we run LLM-guided evolution to discover faster practical algorithms tailored to the task. **Right.** The method discovers algorithms faster than established baselines on a variety of tasks, for example, where we vary the target accuracy level for large ill-conditioned matrices with size $(20000, 2000)$ and conditioning 10^6. The best baseline changes with the task, but our framework adapts to the task and can discover tailored improvements that are especially large in low-accuracy regimes.

Human researchers are limited by time and domain knowledge and hence often do not spend suitable amounts of time optimizing the tools used. Given the advent of automated LLM-guided algorithm discovery, we argue that investing some time in searching for a better task-specific algorithm can lead to significant runtime improvement for downstream solves. Here, we frame the problem of designing dedicated LS algorithms as a meta-discovery problem (Figure 1): given a process for generating task instances, we leverage offline test-time scaling with verifier feedback information to discover suitable high-performance algorithms.

We demonstrate that this approach is capable of finding runtime improvements across a variety of accuracy levels and across different problem instance families. The cost of the discovery process can later be amortized by using the resulting optimized algorithm to solve the downstream task repeatedly. We thereby move from a worst-case to a *practical-case* perspective on algorithm design for least squares. In a *practical least squares task*, a user has a specific workload that generates instances from a fixed data-generating process. Average-case analysis in which one analyzes an algorithm on random inputs avoids some pitfalls of worst-case analysis, but can lead to poor algorithms for inputs generated by different processes. Therefore, the design of truly practical algorithms must be data-driven, where the resulting algorithm is adapted to instances from the data-generating process of the given task. In summary, we leverage the recent developments in LLM-guided evolutionary search to discover faster practical algorithms tailored to specific applications (Section 3). The main contributions of our work are the following:

- **Worst-case versus practical-case LS algorithm design framework** (Section 3): We introduce an applied perspective to algorithm design and demonstrate how to illuminate the space of least squares algorithms. To the best of our knowledge, this provides the first systematic study of LLM-guided algorithm evolution applied to practical LS problems.
- **Runtime improvement discoveries** (Section 4): Our method discovers faster algorithms for a wide array of synthetic LS problems, real-world tasks and target solution accuracy levels. The efficiency gains are especially strong for large-scale or low-accuracy settings.
- **Generalization & discovery analysis** (Section 5): Our method can be flexibly applied to diverse settings and requires minimal work to customize. Additionally, we demonstrate that the discovered algorithms generalize beyond the meta-training task distribution. Finally, we deconstruct algorithmic improvements and contrast LLM-guided discovery with black-box hyperparameter tuning.

2 Background and Related Work

Least Squares Solvers. Least squares is a well-studied computational problem with several algorithmic families developed to find good solutions. A standard robust solver for (1) when the matrix has full column rank is based on a Householder QR decomposition followed by a triangular solve (Golub, 1965):

$$A = QR$$
$$x^* = R^{-1}Q^\top b$$

Other direct methods are based on pivoted QR, SVD, or Cholesky on the Gram matrix $A^\top A$ (see Björck (2024) for a modern treatment). Sketching can be used to speed up direct methods or preconditioned iterative methods such as conjugate gradient on the normal equations (CGLS, Hestenes & Stiefel, 1952) or LSQR (Paige & Saunders, 1982). The core idea consists of drawing a random matrix $S \in \mathbb{R}^{d\times m}$ with $n \le d < m$, and compressing the information of A into the smaller matrix $A_s = SA$ (Murray et al., 2023; Martinsson & Tropp, 2020). A simplified description of a sketch-and-precondition algorithm running for k iterations with a QR-based preconditioner is the following.

$$\begin{aligned}
&A_s = Q_s R_s \\
&y_0 = Q_s^\top Sb \implies x_0 = R_s^{-1} y_0 \quad \text{(sketch-and-solve solution)} \\
&y_k = \text{LSQR}\left(AR_s^{-1}, b; y_0, k\right) \\
&x_k = R_s^{-1} y_k \quad \text{(sketch-and-precondition solution)}
\end{aligned}$$

Early work on randomized preconditioners by Rokhlin & Tygert (2008) led to faster practical algorithms (Avron et al., 2010; Meng et al., 2014; Cartis et al., 2021). More recent works on adaptive preconditioners (Huh et al., 2026) and backward-stable randomized solvers (Epperly et al., 2026) recommend sparse sketching and warm-starting as defaults for fast algorithms.

LLM-Guided Evolutionary Program Discovery. LLM-guided evolutionary search as a tool for algorithm discovery offers an alternative avenue to traditional genetic programming or Reinforcement Learning for optimizing over the space of algorithms by representing them as computer programs and leveraging LLMs as mutation operators. This is a very active research area popularized by FunSearch (Romera-Paredes et al., 2024) and refined by many works including AlphaEvolve (Novikov et al., 2025), OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2026), and SkyDiscover (Liu et al., 2026). Even more recently, Aït Aoudia et al. (2026) extended these ideas by using coding agents instead of individual API calls. Most applications of LLM-guided search to mathematics and theoretical computer science focus on evaluating the output of the algorithms (e.g., as a certificate for some inequality), rather than properties of the algorithm itself; see Georgiev et al. (2025). AlphaEvolve (Novikov et al., 2025) and AlgoTuner (Press et al., 2025) study tasks where the goal is to optimize the algorithm's runtime, but neither considers general least squares problems.

Algorithm Discovery for Least Squares. A closely related work on data-driven discovery of practical algorithms for least squares is Cho et al. (2025). Their approach leverages Gaussian Process-based black-box tuning to find favorable parameter configurations for a certain family of sketch-and-precondition algorithms. Their approach requires specifying the hyperparameters, categorical variables, and admissible values a priori, and is only studied in the context of randomized iterative algorithms. Generalizing this to fully capture the space of algorithms for least squares and the available hardware/software details is laborious and very difficult to achieve. Our approach provides a flexible alternative: algorithms are represented as computer programs and evolved using frontier LLMs.

More recently, Xiong et al. (2026) used reinforcement learning to discover symbolic programs from a specified curriculum of increasingly difficult tasks. Their method succeeds in re-discovering known high-level algorithms, which highlights the interpretability of symbolic program discovery. Our settings are different in that we do not specify the prescribed sequence of algorithmic properties via a curriculum, and we use LLM-guided evolution to find a faster practical algorithm tailored to the task of interest. We will see that the algorithms discovered by our framework are also interpretable.

3 TASK & DISCOVERY METHODOLOGY

Least Squares Algorithm Discovery Tasks. We define a *least squares task* as the tuple $\mathcal{T} = (I_{\text{train}}, I_{\text{test}}, \varepsilon, \texttt{msg})$, where $I_{\text{train}} = \{((A_1, b_1), x_1^*), ..., ((A_K, b_K), x_K^*)\}$ and I_{test} are sets of least squares data instances, $\varepsilon > 0$ is the accuracy target and `msg` is a natural language description of the task. I_{train} is used during the meta-discovery phase to "train" an algorithm for $\mathcal{T}$, while I_{test} acts as a set of held-out instance-solution pairs, which are used to study the generalization quality of the discovered algorithm. Throughout this paper, we focus on dense, full-rank and inconsistent (i.e. $\|Ax^* - b\| > 0$) least squares problems. There are different notions of accuracy for a solution to a least squares problem as in (1). In this paper, we follow the standard $(1 + \varepsilon)$-approximation criterion in randomized numerical linear algebra (Clarkson & Woodruff, 2017) and consider the relative residual ratio:

$$f_{\text{res}}(x) = \frac{\|Ax - b\|_2}{\|Ax^* - b\|_2}. \tag{2}$$

We say that an algorithm $\mathcal{A}$ is feasible for solving the task $\mathcal{T}$ if the output of $\mathcal{A}$ on all instances of $\mathcal{T}$ satisfies $f_{\text{res}} \leq 1 + \varepsilon$.

LLM-Guided Program Evolution Framework. We explore the space of algorithms for a given least squares task $\mathcal{T}$, which is a large and non-differentiable search space. Two components are needed to successfully optimize over this space: a way to effectively explore it by generating candidate algorithms **(C1)**, and a way to accurately and robustly score a candidate algorithm **(C2)**.

Traditionally, this search was done manually by human hypothesis testing. Here, we aim to automate the design of suitable algorithms based on automated iterative program refinement. Due to their strong coding and mathematical reasoning capabilities, modern LLMs offer a very interesting black-box sampling oracle for component **(C1)**. The first natural thing to try is to just sample many times from (an ensemble of) LLMs and keep the best program (*best-of-N*). A different approach is to use test-time scaling frameworks inspired by evolutionary computation to balance exploration with exploitation of different ideas already attempted. We adopt the latter (Lange et al., 2026) but validate other alternatives (see Appendix C). The initial program used for every task presented in this paper was always the same: a randomized preconditioned iterative algorithm using a sparse sketch to build the preconditioner followed by LSQR. The task description "`msg`" was essentially fixed, with minimal changes describing the desired accuracy target, see Appendix B for more details.

The natural candidate for component **(C2)** is the typical runtime of the algorithm on the target matrices, conditional on the output of the algorithm achieving sufficient accuracy. This poses several challenges, because measuring program runtime directly often gives a very noisy estimate of true runtime, due to the state of the machine at the moment of running (e.g. contention, thermal state, cold vs warm cache, etc.).

The Least-Squares Discovery Objective Function. Let $\mathcal{A}$ be the algorithm currently being evaluated, $\mathcal{A}_0$ be the initial algorithm, and $\mathcal{T}$ be the task of interest as defined in Section 3. For training matrices (A_i, b_i) in $\mathcal{T}$, we run algorithms $\mathcal{A}$ and $\mathcal{A}_0$ R times, and let $t_{\mathcal{A}}(i, r)$ be the runtime of $\mathcal{A}(A_i, b_i)$ on run r. Let $\rho(i, r) = \frac{t_{\mathcal{A}}(i,r)}{t_{\mathcal{A}_0}(i,r)}$ be the ratio of the runtime of the current program over the starting one, then we define the *log geometric-mean speedup* of $\mathcal{A}$ as

$$f(\mathcal{A}) = -\frac{1}{K} \sum_{i=1}^{K} \log_2 \left(\text{median}_{r=1,...,R}(\rho(i, r))\right) \tag{3}$$

Focusing on the ratio ρ relative to a fixed algorithm instead of raw runtime t has two benefits. First, the normalization provides a scale-invariant objective, which gives each instance equal weight in terms of multiplicative speedup over $\mathcal{A}_0$. Second, re-timing the initial program each time lowers the fluctuations due to machine contention at each generation. Similarly, running an algorithm R times has two benefits. First, many practical least squares algorithms are randomized, so we can increase our confidence in the algorithm's correctness by running multiple trials. Second, taking the median

among restarts reduces the timing noise.[1] More details on other measures taken to reduce runtime noise are in Appendix B.

Some candidate algorithms might fail to reach the target accuracy ε on all matrices and all runs. A simple way to deal with incorrect programs is to assign score $-\infty$, but this is too non-smooth and does not allow close-to-working ideas to survive. We say that an instance (A_i, b_i) is "solved" by $\mathcal{A}$ if all R runs reach the desired accuracy $f_{\text{res}} \leq 1 + \varepsilon$, and "unsolved" otherwise, and we add a soft penalty that gracefully degrades with the number of unsolved instances. Let S be the subset of solved instances, $\tau < 0$ be a parameter demarcating the threshold between correct and incorrect algorithms, and $g(\mathcal{A}, S) > 0$ be a penalty function defined in Appendix B. Consider the score function

$$\text{score}(\mathcal{A}) = \begin{cases} \max\{f(\mathcal{A}), \tau\} & \text{if } |S| = K \\ \tau - g(\mathcal{A}, S) & \text{if } |S| < K \end{cases} \tag{4}$$

Our discovery framework combines **(C1)** and **(C2)** by using ShinkaEvolve (Lange et al., 2026) to generate algorithms $\mathcal{A}$ with evaluation function (4). The only information from $\mathcal{T}$ that is exposed to the LLM are `msg`, ε and some aggregate information from I_{train} such as average instance size or conditioning, while the full instances and solutions inside I_{train} are used privately by the evaluator to compute the score (4). I_{test} is only used for held-out testing.

The total computational cost of the discovery framework consists of $2RK$ least squares solves per candidate evaluation, excluding the cost of collecting the instances in I_{train} and computing their reference solutions. The exact time required varies depending on the configuration chosen and the complexity of the task considered, and the amount of time waiting for a response from the LLM queries. For example, large-scale, high-accuracy tasks take longer to be solved so they make the whole discovery pipeline slower. Moreover, if someone has a machine with very controlled minimal contention, they can speed up evaluation by removing co-timing or reducing the number of repeats. As a representative, the tasks discussed in Figure 3 below required on average around 3 hours to produce the final algorithm, using around \$10 in API spend. The dollar cost was pretty consistently around that amount on all experiments discussed below, because it depends mostly on the number of generations, which is fixed to $N = 200$.

4 Discovering Practical LS Algorithms

Experimental Setup & Baselines. Our discovery framework supports a variety of hardware and programming languages, but in this paper we focus on CPU and Python (see Appendix A for more details). We generate synthetic instances $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ with $\|A\|_2 = \|b\|_2 = 1$ as follows. First, choose a condition number $\kappa \geq 1$ and fix the optimal residual value to $r^* = 0.01$.

1. Sample two random matrices with orthonormal columns $U \in \mathbb{R}^{m \times n}$ and $V \in \mathbb{R}^{n \times n}$.
2. Generate a list of singular values $1 = \sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n \geq 0$ decaying exponentially as $\sigma_i = \kappa^{-\frac{i-1}{n-1}}$, and set $\Sigma = Diag(\sigma)$. Set $A = U\Sigma V^\top$.
3. Let $w \in \mathbb{R}^n$ be a random vector on the unit sphere, u be a unit vector orthogonal to the column space of A, and set $b = r^* u + \sqrt{1 - (r^*)^2} Uw$. This implies that $\|b\| = 1$, the optimal residual is r^*, and $x^* = \sqrt{1 - (r^*)^2} V\Sigma^{-1} w$.[2]

We consider several synthetic tasks, varying different parameters such as matrix size, condition number, and target accuracy. We compare the discovered programs with several state-of-the-art baselines (Appendix D):

- **QR**: Householder-QR decomposition based on the LAPACK driver "`gels`", which is the robust full-rank reference solver for the task.

[1] The minimum is a popular choice of aggregator for dealing with runtime noise, because contention can only make programs slower. We choose the median because some algorithms are randomized and it removes outliers and avoids selecting lucky seeds.

[2] This ensures Ax^* is isotropic on the column space of A, so iterative algorithms do not have unusually fast convergence due to alignment between solution and singular vectors. As usual, there will be some floating point error in the stored x^* that degrades with κ.

- **Cholesky**: Direct solver for the normal equations based on the Cholesky decomposition of the Gram matrix, which is the fast direct baseline.
- **Blendenpik** (Avron et al., 2010): We set the embedding dimension $d = 2n$ and warm-start the iterative algorithm to the sketch-and-solve initialization.
- **LSQRsparse**: A sparse sketch followed by preconditioned LSQR. This replaces the sub-sampled trigonometric sketching operator used in Blendenpik with an OSNAP sketch of dimension $d = 3n$ and column sparsity $s = 8$ (Nelson & Nguyên, 2013). The initial program used for all tasks is very similar to this baseline (Appendix B).

Throughout all our synthetic tasks, we use 20 instances in the training phase, and we report the average runtime of the discovered algorithm on a disjoint set of 10 test instances.[3]

As a real-world application, we also apply our discovery framework to the SUSY dataset (Whiteson, 2014), which is used as a benchmark to detect exotic particles from high-energy collisions in particle physics (Baldi et al., 2014). We consider the kernel regression problem following Epperly (2024) with $n = 2000$ features. This is a naturally dense tall least squares problem, and discovering faster tailored algorithms might help researchers classify more collisions efficiently.

Single-Task LS Discovery Results. The first synthetic task consists of matrices with size $m = 30000$ and $n = 2000$, condition number $\kappa = 10^6$, and target accuracy $\varepsilon = 10^{-4}$. This is a relatively large ill-conditioned task which needs to be solved with medium accuracy. During the exploration phase, the LLM-guided framework proposes and evaluates many programs, with fast improvement in the first few iterations but also sustained progress in the late stages (see Figure 2, left). The discovered algorithm is a sketch-and-precondition iterative algorithm, but differs substantially from the initial one, and it is $1.42\times$ faster than the best baseline on the test set (see Figure 2, right). It uses a sparse sketching operator of dimension $3n$ and 1 non-zero per column, followed by a Cholesky preconditioner which falls back to QR if the Gram matrix loses positive definiteness, and then preconditioned Conjugate Gradient (CGLS) instead of LSQR, together with many low-level optimizations. Moreover, the stopping rule also differs from the one typical in CGLS (Appendix F).

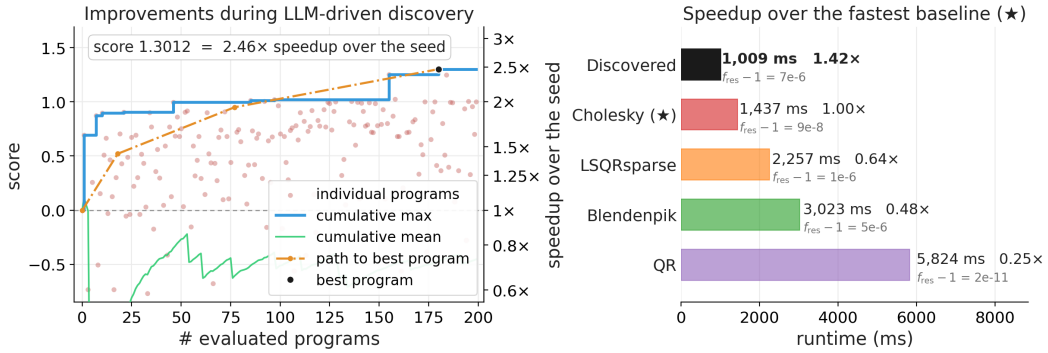

Figure 2: **Left.** Evolution of the search performance scores during the LLM-guided discovery phase. We observe initial rapid runtime performance improvements with sustained minor tuning. **Right.** Performance of the discovered algorithm against the baselines. The discovered sketch-and-precondition algorithm outperforms all considered baselines, with $1.42\times$ to $5.77\times$ speedup.

Multi-Task LS Discovery Results. To study the flexibility of our method, we considered a grid of neighboring tasks, where we vary matrix size and condition number, with $(m, n) \in \{(20000, 1000), (20000, 2000), (30000, 2000)\}$ and $\kappa \in \{10^2, 10^6, 10^{10}\}$. The best baseline for all tasks considered here with $\kappa < 10^{10}$ is always Cholesky on the normal equations, although this is not a robust solver because it fails to reach the desired accuracy at large condition numbers. The best baseline for ill-conditioned matrices is the randomized preconditioner from a sparse sketch. Our framework discovers faster algorithms for all these tasks, which are up to $1.73\times$ faster than the best

[3]We omit the error bars because all runtimes are within $1 - 3\%$ of the average for all synthetic tasks.

baseline for size $(20000, 2000)$ and condition number 10^2. This is shown in Figure 3, together with the performance of the program discovered at size $(30000, 2000)$ and condition number 10^6. This shows that the discovered algorithms can be empirically robust and perform better than the baselines across different tasks.

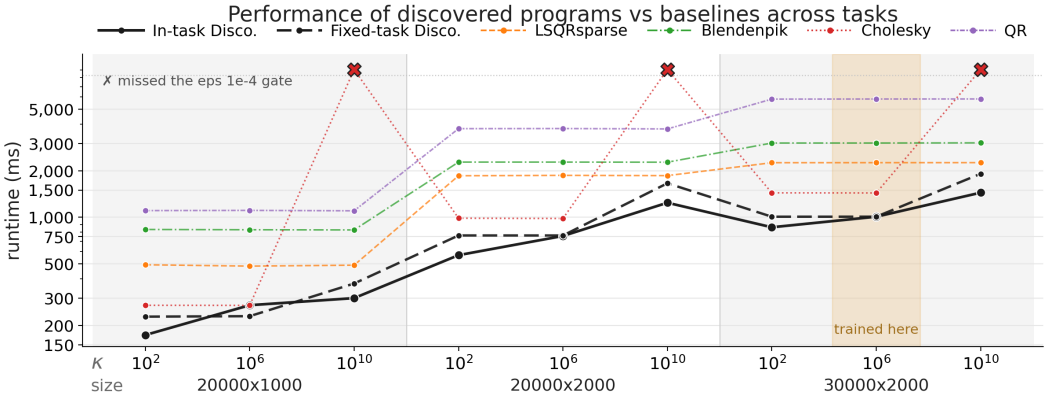

Figure 3: Test-set performance comparison between baselines and in-task discovered algorithms (solid black line) and a fixed discovered algorithm (dashed black line), across a variety of tasks. Our framework discovers comparable or faster algorithms across all tasks, and these algorithms generalize well outside the task they were trained on.

As a third synthetic experiment, we fix matrix size $(20000, 2000)$ and condition number 10^6, and vary the target accuracy level $\varepsilon \in \{1, 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\}$. Again, we observe a transition where direct Cholesky is the fastest baseline when the task solvability is far from the accuracy floor, and then fails as the required accuracy is tighter. The discovered algorithms tie or beat every baseline in every task again, as shown in Figure 1.

Real-World LS Discovery Results. Beyond the previously studied synthetic tasks, we investigate the usefulness of our proposed workflow in an applied real-world setting. Using the SUSY dataset (Whiteson, 2014) for high-energy particle physics, we form kernel regression least squares tasks by selecting 10 disjoint blocks of $100,000$ rows used for training, which correspond to matrices of size $100,000 \times 2000$. We then evaluate the algorithm on three disjoint sets of $1,000,000$ rows. Since the exact minimizer x^* is not known for these instances, we use the result of a QR direct solve as a reference. On a medium accuracy task $\varepsilon = 10^{-4}$, our method discovers an algorithm which finds a solution $2\times$ faster than the next best baseline, while on a high accuracy task $\varepsilon = 10^{-12}$ the speedup is $1.52\times$, see Table 1 for more details.

Table 1: Performance of discovered algorithms on instances of $1,000,000$ rows from the SUSY dataset. The error $f_{\text{res}} - 1$ is computed relative to the reference solution found by the QR direct solver, hence the 0 in the corresponding row. The reported error and runtime are the averages over the test set, and Cholesky fails to reach the target accuracy for $\varepsilon = 10^{-12}$ (red).

	$\varepsilon = 10^{-4}$			$\varepsilon = 10^{-12}$		
Program	time (s)	speedup	error	time (s)	speedup	error
Discovered	**20.7**	$2.00\times$	2.4×10^{-5}	**48.3**	$1.52\times$	1.1×10^{-15}
LSQRsparse ($\star$)	41.4	$1.00\times$	9.4×10^{-7}	73.7	$1.00\times$	4.6×10^{-14}
Blendenpik	94.0	$0.44\times$	3.4×10^{-6}	145.4	$0.51\times$	1.1×10^{-13}
Cholesky	45.6	$0.91\times$	8.2×10^{-11}	45.4	–	8.2×10^{-11}
QR	242.7	$0.17\times$	0^*	243.2	$0.30\times$	0^*

5 Analysing the Discovered LS Algorithms

Next, we provide an in-depth study of the algorithms discovered by our methodology, highlighting the sources of efficiency gains. Moreover, we compare our LLM-guided discovery framework with hyperparameter tuning, showing that LLM search can outperform hyperparameter tuning, but the two can be combined to find even faster algorithms.

Algorithm Analysis for a Fixed Task. We consider the algorithm discovered in the task with matrices of size $(30000, 2000)$, condition number 10^{10}, and accuracy target 10^{-4}. This remains a randomized preconditioner, but most pieces of the initial program were changed, including the termination condition on the backward error (Δ_1), the sketch family (Δ_2), the sketch dimension (Δ_3), together with some low-level code optimization (Δ_4). See Figure 4 for a detailed breakdown of each contribution, while we defer the full description of each program to Appendix F, which also contains the analysis of algorithms discovered for different tasks. One particularly interesting discovery is a randomized preconditioner with a new stopping criterion which is mathematically sound and provides a speedup over the traditional backward error rule (see Appendix F, Eq. 6).

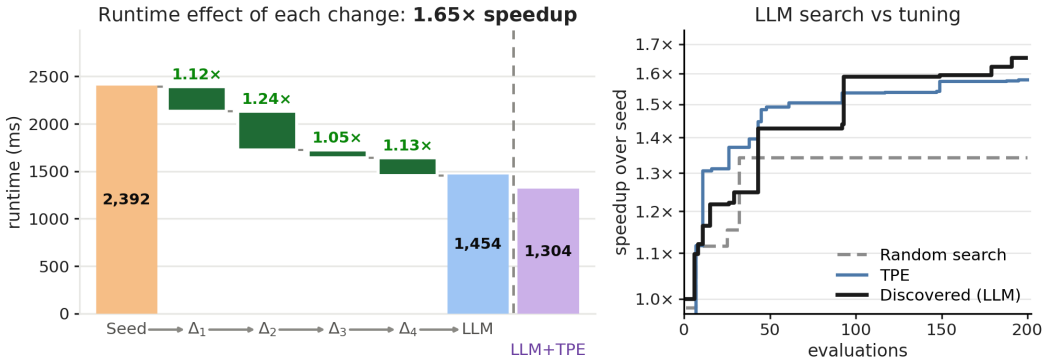

Figure 4: **Left.** Breakdown of the runtime improvement achieved by the discovered algorithm. Δ_1 changed the termination condition *atol* from ε to $0.16\sqrt{\varepsilon}$, Δ_2 changed the sketching family from OSNAP with 8 nonzeroes per column to row-sampling with row-norms probabilities, Δ_3 lowered the sketch dimension from $3n$ to $1.9n$, and Δ_4 includes some code optimizations. Performing hyperparameter tuning via TPE on the discovered algorithm can further improve its performance by $1.11\times$. **Right.** Comparison between LLM-guided discovery (black) and hyperparameter optimization starting from the same initial program for this task. We considered two hyperparameter search methods: 200 iterations of TPE (blue) or taking the best of 200 random valid configurations (grey).

Program Evolution versus Hyperparameter Optimization. Comparing arbitrary program evolution with hyperparameter optimization is generally delicate because many types of mutations are difficult to represent as hyperparameters. However, many randomized iterative algorithms for least squares naturally lend themselves to it (Cho et al., 2025), so we compared our framework with random hyperparameter search and more advanced hyperparameter tuning using the tree-structured Parzen estimator (TPE, Bergstra et al., 2011) implemented via Optuna (Akiba et al., 2019) on the task of the previous paragraph. Our framework outperformed both random search and TPE, and it kept producing improvements at later stages of the exploration. We also considered combining the two approaches, where we run a few iterations of TPE starting from the program discovered by our framework, and this led to even faster algorithms; see Figure 4. This suggests that LLM-guided evolution can make larger "jumps" in late iterations, which helps prevent search stagnation typical of Bayesian hyperparameter tuning, but the local optimization provided by hyperparameter tuning can further refine these solutions. However, not all discovered algorithms might naturally lend themselves to hyperparameter tuning, for example direct algorithms do not expose a comparable set of tunable hyperparameters as randomized iterative methods. Another drawback of hyperparameter tuning is that the search space is constrained by a user-specified family of algorithms defined by the hyperparameters chosen, while LLM-guided search is unrestricted and much more flexible. The hyperparameter configurations considered in the search are in Appendix E.

Program Evolution versus Random Sampling. We compare our framework based on LLM-guided evolutionary search and a *best-of-N* LLM sampling strategy. Both methods achieve a significant improvement over the initial program, but the evolutionary approach used by our framework yields a $1.16\times$ speedup over random sampling from LLMs, thanks to the deeper exploitation of previously proposed ideas (Figure 5, right). Detailed results are provided in Appendix C.

Out-of-Distribution Generalization. To understand the out-of-distribution generalization capabilities of the algorithms discovered by our framework, we consider a task with a mix of two families of instances evenly split between $((m, n), \kappa) \in \{((20000, 1000), 10^{10}), ((30000, 2000), 10^{2})\}$. We evaluate the discovered algorithm over a grid of instances with different size and condition number, and different target accuracy levels (Figure 5, left). Runtime speedups transfer strongly across size and conditioning, where the algorithm outperforms all baselines, but the target accuracy generalizes only on the looser side, with the algorithm failing to reach the target accuracy at $\varepsilon = 10^{-6}$.

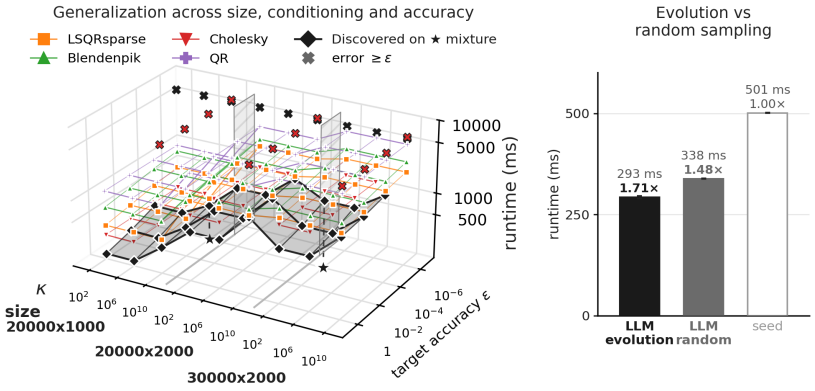

Figure 5: **Left.** Generalization of a discovered LS algorithm across a grid of different size, conditioning, and accuracy targets. The algorithm is trained at accuracy level 10^{-4} and a mixture of instances of size and conditioning moving in opposite directions, and its generalization is one-sided, working well at lower accuracy but failing at higher accuracy. The left axis flattens the size and condition number axes, making it possible to visualize this 4-dimensional data in 3D. **Right.** Comparison between our framework (LLM evolution) and best-of-N sampling (LLM random) on a task with size $(20000, 1000)$, $\kappa = 10^{10}$, $\varepsilon = 10^{-4}$, yielding $1.71\times$ and $1.48\times$ speedup over the seed.

6 Conclusion

Summary. We introduced a framework for using LLM-guided search to discover faster algorithms tailored to practical least squares applications. Our results demonstrate significant runtime speedups over robust baselines and default implementations with minimal task-specific input, offering a way to automate the tedious hand-tuning required to speed up stable algorithms for LS tasks.

Limitations. Our framework requires pre-computing solutions to some instances and repeatedly solving many least squares problems during evaluation, so the benefits of the faster discovered algorithms become tangible in applications which require several downstream or repeated solves. The algorithms discovered are often a recombination of LS techniques known to experts. Finding a genuinely novel high-level algorithmic discovery remains challenging. The algorithms discovered lack provable guarantees beyond their success on task instances, so they should be carefully inspected.

Future Directions. Given the flexibility of our framework, it would be very interesting to apply it to other tasks beyond dense least squares, for example to sparse settings or to other linear algebraic problems. Further understanding and improving the LLM-guided exploration framework to discover new provably stable algorithms is also an interesting research direction.

Broader Impact & Ethical Considerations. We hope our work can be useful to practitioners who need to quickly solve many large-scale least squares problems. We will release an open-source repository with our methodology and the algorithms discovered for the tasks considered in this paper.

AI USE STATEMENT

In this work, we used generative AI tools for writing code used for generating synthetic tasks and implementing the experiments, assistance in interpreting the results, and producing the figures. We have not used generative AI tools for designing or providing feedback on research methodology or experiments, and writing the paper, aside from grammar checks and formatting. Other tasks with required disclosure are not applicable to this work. Many algorithms discussed in this paper were produced with the assistance of generative AI tools, but that is a central component of the research methodology, and it is discussed in detail in the paper.

We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness, and all data presented in the figures were verified to be consistent with the results of the experiments. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

To the best of our knowledge, our work on discovering faster practical algorithms for least squares does not present major ethical or societal hazards. The research conducted in this work uses closed frontier models which limit the ability of researchers with financial constraints to use it repeatedly at scale. However, the framework can be easily adapted to use cheaper or open-weight models, by changing the LLM-guided evolutionary algorithm model APIs.

REPRODUCIBILITY STATEMENT

Program evolution based on LLM-guided mutations is not perfectly reproducible because of LLM API calls, which cannot be reproduced by fixing a "seed". However, we observed that this framework can consistently produce improvements over the baselines, so even if the exact programs might not be perfectly reproducible, the overall effect is empirically consistent. We plan to make all the code used for the experiments in this paper available in an open-source repository. All least squares instances are reproducible, but exact runtimes are not because they depend on software, hardware, and state of the machine at the time of execution.

REFERENCES

Fayçal Aït Aoudia, Jakob Hoydis, Sebastian Cammerer, Lorenzo Maggi, Gian Marti, and Alexander Keller. The AI telco engineer: Toward autonomous discovery of wireless communications algorithms. *arXiv preprint arXiv:2604.19803*, 2026.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

Edward Anderson, Zhaojun Bai, Christian Bischof, L. Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK users' guide*. SIAM, 3 edition, 1999.

Mario Arioli and Iain S. Duff. Preconditioning linear least-squares problems by identifying a basis matrix. *SIAM Journal on Scientific Computing*, 37(5):S544–S561, 2015.

Richard C. Aster, Brian Borchers, and Clifford H. Thurber. *Parameter estimation and inverse problems*, volume 90. Elsevier academic press Amsterdam, 2005.

Haim Avron, Petar Maymounkov, and Sivan Toledo. Blendenpik: Supercharging LAPACK's least-squares solver. *SIAM Journal on Scientific Computing*, 32(3):1217–1236, 2010.

Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):4308, 2014.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

Philip R. Bevington and D. Keith Robinson. *Data reduction and error analysis for the physical sciences*. McGraw-Hill New York, 3 edition, 2003.

Åke Björck. *Numerical methods for least squares problems*. SIAM, 2 edition, 2024.

Coralia Cartis, Jan Fiala, and Zhen Shao. Hashing embeddings of optimal dimension, with applications to linear least squares. *arXiv preprint arXiv:2105.11815*, 2021.

Younghyun Cho, James Demmel, Michał Dereziński, Haoyun Li, Hengrui Luo, Michael Mahoney, and Riley Murray. Surrogate-based autotuning for randomized sketching algorithms in regression problems. *SIAM Journal on Matrix Analysis and Applications*, 46(2):1247–1279, 2025.

Kenneth L. Clarkson and David P. Woodruff. Low-rank approximation and regression in input sparsity time. *Journal of the ACM (JACM)*, 63(6):1–45, 2017.

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

Ethan N. Epperly. Fast and forward stable randomized algorithms for linear least-squares problems. *SIAM Journal on Matrix Analysis and Applications*, 45(4):1782–1804, 2024.

Ethan N. Epperly, Maike Meier, and Yuji Nakatsukasa. Fast randomized least-squares solvers can be just as accurate and stable as classical direct solvers. *Communications on Pure and Applied Mathematics*, 79(2):293–339, 2026.

Eric D. Feigelson and G. Jogesh Babu. *Modern statistical methods for astronomy: with R applications*. Cambridge, 2012.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. AI and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

Gene Golub. Numerical methods for solving linear least squares problems. *Numerische mathematik*, 7(3):206–216, 1965.

Gene H. Golub and Charles F. Van Loan. *Matrix computations*. JHU press, 4 edition, 2013.

Per Christian Hansen, Jakob Jørgensen, and William R.B. (eds.) Lionheart. *Computed tomography: algorithms, insight, and just enough theory*. SIAM, 2021.

Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

Jung Eun Huh, Coralia Cartis, and Yuji Nakatsukasa. Adaptive LSQR preconditioning from one small sketch. *arXiv preprint arXiv:2604.05065*, 2026.

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.

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, et al. SkyDiscover: A flexible, adaptive framework for AI-driven scientific and algorithmic discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 1223–1227, 2026.

Per-Gunnar Martinsson and Joel A. Tropp. Randomized numerical linear algebra: Foundations and algorithms. *Acta Numerica*, 29:403–572, 2020.

Xiangrui Meng, Michael A. Saunders, and Michael W. Mahoney. LSRN: A parallel iterative solver for strongly over- or underdetermined systems. *SIAM Journal on Scientific Computing*, 36(2): C95–C118, 2014.

Riley Murray, James Demmel, Michael W. Mahoney, N. Benjamin Erichson, Maksim Melnichenko, Osman Asif Malik, Laura Grigori, Piotr Luszczek, Michał Dereziński, Miles E. Lopes, Tianyu Liang, Hengrui Luo, and Jack Dongarra. Randomized numerical linear algebra: A perspective on the field with an eye to software. *arXiv preprint arXiv:2302.11474*, 2023.

Jelani Nelson and Huy L. Nguyên. OSNAP: Faster numerical linear algebra algorithms via sparser subspace embeddings. In *2013 IEEE 54th annual symposium on foundations of computer science*, pp. 117–126. IEEE, 2013.

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

Christopher C. Paige and Michael A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software (TOMS)*, 8(1):43–71, 1982.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, et al. AlgoTune: Can language models speed up general-purpose numerical programs? *Advances in Neural Information Processing Systems*, 38, 2025.

Vladimir Rokhlin and Mark Tygert. A fast randomized algorithm for overdetermined linear least-squares regression. *Proceedings of the National Academy of Sciences*, 105(36):13212–13217, 2008.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J.R. 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.

Alessandro Rudi, Luigi Carratino, and Lorenzo Rosasco. FALKON: An optimal large scale kernel method. *Advances in neural information processing systems*, 30, 2017.

Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

Asankhaya Sharma. OpenEvolve: an open-source evolutionary coding agent. *GitHub repository https://github.com/algorithmicsuperintelligence/openevolve*, 2025.

Joel A. Tropp. Improved analysis of the subsampled randomized Hadamard transform. *Advances in Adaptive Data Analysis*, 3(01n02):115–126, 2011.

Eric Vittinghoff, David V. Glidden, Stephen C. Shiboski, and Charles E. McCulloch. *Regression methods in biostatistics: linear, logistic, survival, and repeated measures models*. Springer, 2005.

Daniel Whiteson. SUSY. UCI Machine Learning Repository, 2014. DOI: https://doi.org/10.24432/C54606.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Jeffrey M. Wooldridge. Introductory econometrics: A modern approach. 2009.

Jinglong Xiong, Xiaotian Liu, Ruoxin Wang, Zihang Liu, Yefan Zhou, Yujun Yan, and Yaoqing Yang. RL4RLA: Teaching ML to discover randomized linear algebra algorithms through curriculum design and graph-based search. *arXiv preprint arXiv:2605.18004*, 2026.

APPENDIX TABLE OF CONTENTS

A Hardware and Software Packages **14**

B More details on the methodology **14**

B.1 Evaluation function 14

B.2 ShinkaEvolve parameters 15

C Tradeoff between LLM-guided evolution and random sampling **16**

D Baselines **17**

E Hyperparameter tuning settings **19**

F Further analysis of discovered algorithms **19**

G Other ideas explored **22**

G.1 Deterministic robust proxy for runtime 22

G.2 Minimizing residual error subject to a time budget 23

H Discovered Practical Programs for Least Squares **23**

A HARDWARE AND SOFTWARE PACKAGES

All experiments were performed on a shared node inside a SLURM-managed partition with 2TB of DDR5 memory and Intel Xeon Platinum 8481C 2.7GHz CPU with 104 cores. Up to 60 cores were used during the LLM-guided search phase to parallelize candidate evaluation, but all timings on test instances presented in the paper were measured with exclusive use of the node and 32GB of memory. We used CPython 3.14.5 with NumPy 2.4.6, SciPy 1.17.1, OpenBLAS 0.3.30, and ShinkaEvolve 0.0.7.

B MORE DETAILS ON THE METHODOLOGY

B.1 EVALUATION FUNCTION

The log geometric-mean speedup function (3) is designed to reduce runtime noise (hence the median) but also preserve signals coming from variations across instances (hence the mean).

Let $S \subseteq I_{\text{train}}$, and define the restriction of (3) on S as

$$f(\mathcal{A}; S) = -\frac{1}{|S|} \sum_{i \in S} \log_2 \left(\text{median}_{r=1,\ldots,R}(\rho(i, r))\right)$$

and let $\sigma(x) = \frac{1}{1+e^{-x}}$ be the sigmoid function. Define the infeasible scores for (4) as

$$g(\mathcal{A}, S) = \begin{cases} K - |S| - \sigma(f(\mathcal{A}; S)) & \text{if } |S| > 0 \\ K - |S| & \text{if } |S| = 0 \end{cases}$$

This objective function was chosen so that programs with more correct instances are always better than programs with fewer ones, and speed differentiates programs that solve the same number of instances. The floor τ is added to separate the feasible and infeasible region. Throughout all our experiments, we fix $\tau = -5$ and $R = 3$.

The function (4) is designed so that a feasible initial program $\mathcal{A}_0$ always has score 0, with some tiny fluctuations due to runtime measurement noise, while programs with score greater than 0 are always feasible and faster. The score function is chosen so that score$(\mathcal{A}) = 1$ means that the algorithm $\mathcal{A}$ takes half the time as $\mathcal{A}_0$, score$(\mathcal{A}) = -1$ means double the time, and in general score$(\mathcal{A}) = s$ means a 2^{-s} fraction of the time.

To reduce runtime noise, we adopt the following measures.

- We normalize the state of the machine/caches by running a fixed program before any timed evaluation.
- We interleave the initial program and the current program evaluations, i.e. we compute $t_{\mathcal{A}}(1,1), t_{\mathcal{A}_0}(1,1), t_{\mathcal{A}}(1,2), t_{\mathcal{A}_0}(1,2), ..., t_{\mathcal{A}}(K,3), t_{\mathcal{A}_0}(K,3)$ in this order.
- We pin processes used in the evaluation to physical cores.

B.2 SHINKAEVOLVE PARAMETERS

All the experiments in this paper used the same initial program (Listing 2) and the same configuration of ShinkaEvolve parameters (Table 2). All runs completed 200 generations using around $10 per task despite the larger cost budget, but one can lower the cost even further by using a cheaper LLM ensemble.

Listing 1: Task `msg` parameter used as `task_sys_msg` inside ShinkaEvolve. The only part that changes across different tasks is the one on the target accuracy, where we report the desired value and label $\varepsilon \in \{1, 0.01\}$ as "low-precision", $\varepsilon \in \{10^{-4}, 10^{-6}\}$ as "moderate-precision", and $\varepsilon \in \{10^{-8}, 10^{-10}\}$ as "high-precision".

```
"""
You are an expert in numerical linear algebra and randomized algorithms.

TASK: write ONE Python function that solves the overdetermined least-
    squares problem  min_x ||A x - b||_2
(A is a real numpy array of shape (m, n) with m >> n; b is a real numpy
    vector of shape (m,)) as FAST as possible in measured wall-clock time
    .

`eps` is the target residual accuracy ||A x - b|| <= (1 + eps) * min_x ||
    A x - b||.
For this task, we are interested in moderate-precision solutions, so eps
    is on the order of 1e-4.  However, it is a RUN-TIME PARAMETER, not a
    constant, and you cannot check that condition without solving the
    problem, so DECIDING WHEN TO STOP IS PART OF THE PROBLEM.

Build a stopping test out of what you can actually see -- A, b, your
    current iterate -- and make it cheap, because whatever it costs is
    timed, but also accurate enough (failing to solve the problem to the
    desired accuracy results in a large penalty).

You receive input data in double-precision floats, but feel free to try
    lower precision arithmetic to make the algorithm faster.

GOAL: minimize the wall-clock time to a solution that reaches the
    accuracy on EVERY problem. Be fast AND dependable -- a program that
    fails an instance, or that is occasionally far slower than usual, is
    not good enough.

Feel free to use optimized linear algebra libraries like numpy and scipy.
Try sketching or a combination of randomized and direct methods in any
    way you see fit.

Because the objective is REAL wall-clock time, cheap-but-good beats
    theoretically-optimal: prefer whole-array numpy BLAS-3 operations
    when possible, avoid recomputing, and minimize total time to an
    accepted x.
```

```
"""
```

Listing 2: Initial program used by our discovery framework

```python
# EVOLVE-BLOCK-START
import numpy as np
from scipy import linalg as sla
from scipy.sparse import csr_matrix
from scipy.sparse.linalg import LinearOperator, lsqr

def _osnap(m, d, s):
    cols = np.repeat(np.arange(m), s)
    rows = np.random.randint(0, d, size=m * s)
    signs = np.random.choice((-1.0, 1.0), size=m * s) / np.sqrt(s)
    return csr_matrix((signs, (rows, cols)), shape=(d, m))

def solve(A, b, eps, iter_budget):
    m, n = A.shape

    # --- Preconditioner: sketch A (OSNAP, d=3n rows), thin-QR of the sketch, M = R^{-1} (right-preconditioner
     ).
    d = 3 * n
    S = _osnap(m, d, 8)
    SA = S @ A
    Sb = S @ b
    R = np.linalg.qr(SA, mode="r")
    M = sla.solve_triangular(R, np.eye(n))

    # --- Warm start: sketch-and-solve, so LSQR only iterates on the residual.
    x0 = M @ (M.T @ (SA.T @ Sb))
    b_eff = b - A @ x0
    if np.linalg.norm(b_eff) <= 1e-20:
        return x0

    # --- Preconditioned LSQR on A @ M, stopping on its own backward-error test at tolerance eps.
    op = LinearOperator((m, n), matvec=lambda v: A @ (M @ v),
                        rmatvec=lambda u: M.T @ (A.T @ u), dtype=np.float64)
    y = lsqr(op, b_eff, atol=eps, btol=eps, conlim=0, iter_lim=iter_budget)[0]
    return x0 + M @ y

# EVOLVE-BLOCK-END

# Fixed harness (timing + accuracy check), hidden from the search: it cannot bypass the timing or the grading.

from program_run import run_solve as _run

def run_solve(A, b, nnz, eps, iter_budget, run_idx, r_star, seed):
    return _run(solve, A, b, nnz, eps, iter_budget, run_idx, r_star, seed)
```

C TRADEOFF BETWEEN LLM-GUIDED EVOLUTION AND RANDOM SAMPLING

In this section, we compare the approach described in Section 3 with random sampling from an ensemble of LLMs. We also consider hybrid approaches where we randomly restart k ShinkaEvolve instances of N/k generations each. The extreme cases of $k = 1$ and $k = N$ recover ShinkaEvolve and random LLM sampling, respectively. For settings where we use multiple restarts, we set aside a validation instance set I_{valid} different from I_{train} and I_{test}, which is used to co-time the discovered algorithms from the independent runs and select the best one. This is non-trivial because the score during the evolutionary phase is noisier than during validation and testing due to contention coming from parallel evaluations, so it should not be used to select the best algorithms. In fact, we observed a different validation ranking compared to the training one on multiple occasions.

We fix $N = 200$ throughout and consider $k = 1, 2, 5, 10$, and also consider two versions of random sampling. The first uses `msg` as in Listing 1 and provides the initial program (Listing 2) to the LLM prompt, together with the same public metrics as the evolutionary runs (i.e. this is equivalent to $k = 200$). The second one is labeled "naive" because it provides the most basic seed algorithm (the default `scipy.linalg.lstsq` or direct Cholesky) and more limited context. This was meant to study the effect of prompting on the LLMs' ability to discover fast least squares algorithms.

We ran all discovery regimes on two tasks, one involving large ill-conditioned matrices (size $(20000, 2000)$ and $\kappa = 10^6$), and one involving smaller but even more ill-conditioned matrices (size $(20000, 1000)$ and $\kappa = 10^{10}$), both with target accuracy $\varepsilon = 10^{-4}$. Unsurprisingly, prompting has a strong influence on the exploration abilities of LLMs, as evidenced by the "naive" runs not leaving the initial program on both tasks. On the other hand, all other discovery regimes exhibited substantial improvements over the initial program and the other baselines. For the very ill-conditioned task,

Table 2: ShinkaEvolve parameters that differ from the default ones. The number of generations is 201 because it includes the seed program.

Parameter	Value
`EvolutionConfig`	
`num_generations`	201
`max_api_costs`	20

`llm_models`	`gpt-5.3-codex`
	`gpt-5.6-terra`
	`gpt-5.6-sol`
	`gemini-3.5-flash`
	`gemini-3.1-pro-preview`

`llm_kwargs.reasoning_efforts`	low, medium, high
`llm_kwargs.max_tokens`	64000
`novelty_llm_models`	`gpt-5.6-luna`
`code_embed_sim_threshold`	0.92
`DatabaseConfig`	
`num_islands`	4
`enable_dynamic_islands`	True
`stagnation_threshold`	50
`migration_rate`	0.1
`migration_interval`	40
`archive_size`	80
`archive_selection_strategy`	crowding
`num_archive_inspirations`	4
`num_top_k_inspirations`	2
`parent_selection_lambda`	3
`JobConfig`	
`numeric_threads_per_job`	1
`ShinkaEvolveRunner`	
`max_proposal_jobs`	12
`max_evaluation_jobs`	6
`max_db_workers`	1

all evolutionary regimes performed similarly, with $k = 1$ and $k = 10$ as the best ones. The best regime on the other task was $k = 1$, outperforming $k = 200$ by $1.15\times$ and $k = 2$ by $1.23\times$ (Figure 6). In light of this, we chose the settings with minimal restarts as our default methodology. This also makes the framework simpler and avoids the need for a validation phase to select the best program among the independent restarts. However, one could have chosen other settings described here and would have likely reached similar results, aside from the naive one. Finding more interesting ways to balance exploration with exploitation for optimizing functions over the space of algorithms remains a very interesting and open research question.

D BASELINES

Direct method I: Householder QR The first baseline is the classical stable solver coming from computing $A = QR$ and solving a triangular system $Rx = Q^\top b$, where $Q^\top b$ is computed via Householder reflectors instead of storing Q explicitly. We use the LAPACK driver *dgels* and manually implement this instead of relying on the default `scipy.linalg.lstsq` because that uses an SVD decomposition and it is unnecessarily slow for full-rank least squares problems.

Direct method II: Cholesky on the Gram matrix This is another classical algorithm that forms the normal equations $A^\top Ax = A^\top b$, which is a positive definite linear system when A has full column-rank, so it can be solved by computing the Cholesky decomposition of $G = A^\top A$, i.e.

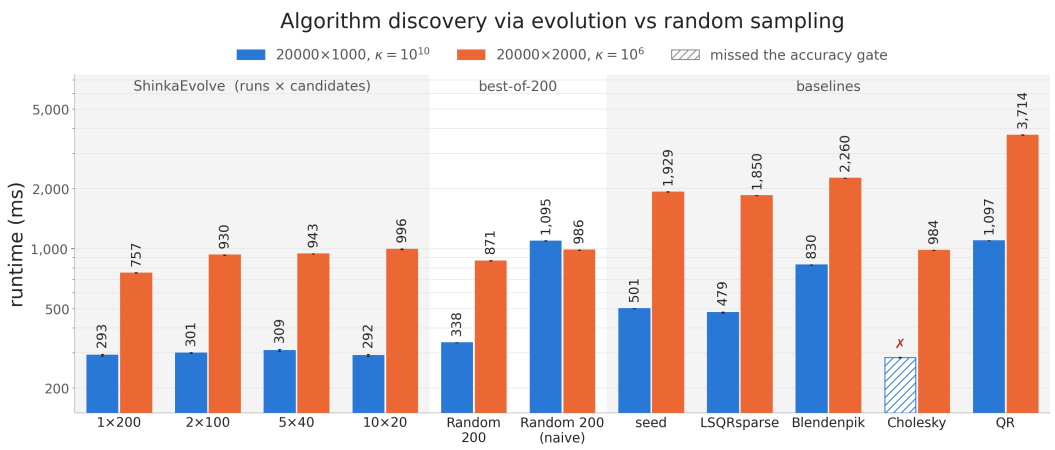

Figure 6: Comparison of performance of algorithms discovered under different settings of program refinement (evolution) and random sampling (restarts). On the tasks considered here, the setting with a single LLM-guided evolutionary run outperformed the other ones, but not by a large margin relative to how much most regimes improve over the baselines. On the regimes with best-of-200, prompting was essential in exploring beyond the initial program. Direct Cholesky failed to reach the target accuracy for the task with condition number 10^{10}.

let $G = LL^\top$ and solve $LL^\top x = A^\top b$ using two triangular solves. This is not as numerically stable as QR, since it computes the Cholesky factor of a matrix with squared condition number $\kappa(G) = \kappa(A)^2$, so it will not solve as many problems as QR to high-precision. However, it is much faster when it is applicable.

Randomized preconditioner I: Blendenpik This algorithm was invented by Avron et al. (2010) and it follows the sketch-and-precondition template from Section 2. The sketching operator $S \in \mathbb{R}^{d\times m}$ is a subsampled randomized Hadamard transform (SRHT)

$$S = \sqrt{\frac{m}{d}} RHD$$

where $D \in \mathbb{R}^{m\times m}$ is a diagonal matrix with entries chosen uniformly in $\{-1, +1\}$, $H \in \mathbb{R}^{m\times m}$ is the scaled Walsh-Hadamard matrix, and $R \in \mathbb{R}^{d\times m}$ is a matrix that restricts an m-dimensional vector to d randomly chosen (but fixed) directions (see Tropp (2011) for more details). The sketch dimension is fixed to $d = 2n$, which makes the algorithm faster than the default recommendation of $d = 4n$ in Avron et al. (2010).

Randomized preconditioner II: sparse sketching This uses the sparse-sign sketching operator of Nelson & Nguyên (2013), where the matrix $S \in \mathbb{R}^{d\times m}$ has $d = 3n$ rows and exactly 8 nonzeroes per column with values chosen independently in $\left\{-\frac{1}{2\sqrt{2}}, +\frac{1}{2\sqrt{2}}\right\}$. The rows are split in 8 disjoint blocks and each nonzero location is chosen at random inside each block. This is a sparse matrix that can be applied to A much more quickly than the dense sketching operator of Blendenpik.

Termination condition for iterative algorithms Both randomized preconditioners use the iterative algorithm LSQR on the preconditioned system $\hat{A} = AR^{-1}$, with R coming from the QR factorization of SA. This iterative algorithm stops when the scaled normal equation residual γ is below a threshold `atol`[4]. Let $\hat{r} = \hat{A}x - b$ be the residual at x, then this termination condition is

$$\gamma = \frac{\|\hat{A}^\top \hat{r}\|}{\|\hat{A}\|\|\hat{r}\|} \leq \texttt{atol}$$

[4]We set the other termination parameter `btol` to 0.

This is an upper bound on the backward error in $\hat{A}$ at x, and using default parameters might not necessarily reflect our target accuracy in the relative residual ratio f_{res}. Therefore, we tune the termination parameter to

$$\texttt{atol} = 0.03\frac{\sqrt{\varepsilon(2+\varepsilon)}}{1+\varepsilon}$$

This termination condition gives a sufficient condition for $f_{\text{res}} \leq 1+\varepsilon$ in exact arithmetic whenever the preconditioned matrix has condition number less than 30, which is the case for all applications in this paper.[5] The term depending on ε is the result of converting a bound on γ to a bound on f_{res}, while the constant 0.03 is fixed to control the bound on the preconditioned condition number, so lowering it extends the guarantee to even more poorly conditioned matrices.

E Hyperparameter tuning settings

The hyperparameter configurations considered in Section 5 are reported in Table 3. Note that this covers a more extensive family of algorithms than the previous work on hyperparameter tuning for randomized preconditioning (Cho et al., 2025). The stopping constant c is set so that the iterative algorithm terminates on backward error $\textit{atol} = c\sqrt{\varepsilon}$, while the row norm/uniform mixing parameter α controls the tradeoff between sampling rows based on their norm or uniformly. The total search space is slightly smaller than the cartesian product of all hyperparameters, because some of them are not compatible with each other. For example, the nonzeros per row or column k is only set when the chosen sketch operator is sparse sign.

The hyperparameters used during search are intended to be broad and cover a large class of algorithms, while the hyperparameters used in the refinement phase are narrowly centered around the discovered solution, i.e. $\gamma = 1.9$, $\alpha = 0.5$, and $c = 0.16$. The refined solution selects sketch size $\gamma = 2.23$, row sampling leverage $\alpha = 0.58$, stopping constant $c = 0.36$.

Table 3: Hyperparameter configurations used for random search and TPE on the least squares task with size $(30000, 2000)$, condition number 10^{10}, and target accuracy 10^{-4}.

Hyperparameter	Allowed values	Sampling
Used during search by TPE and random (200 evaluations)		
Sketch operator	sparse sign (col.), sparse sign (row), row sampling, SRHT	uniform
Sketch size factor γ	$[1,\ 5]$	uniform
Nonzeros per col./row k	$\{1, \ldots, 100\}$, sparse sign only	uniform (int.)
Leverage mixing α	$[0,\ 1]$, row sampling only	uniform
Preconditioner	QR of SA, Cholesky of $(SA)^\top SA$, SVD of SA	uniform
Krylov solver	LSQR, LSMR, CGLS	uniform
Warm start	none, sketch-and-solve	uniform
Stopping constant c	$[0.003,\ 1]$	log-uniform
Used to refine the LLM-discovered programs by TPE (50 evaluations)		
Sketch size factor γ	$[1.2,\ 3.2]$	uniform
Leverage mixing α	$[0.15,\ 0.85]$	uniform
Stopping constant c	$[0.05,\ 0.5]$	log-uniform

F Further analysis of discovered algorithms

This section contains a detailed discussion of some algorithms discovered by our framework together with the full Python programs. We will see that these algorithms can be quite different, ranging from direct to iterative methods, showing that our framework can flexibly adapt to the task at hand and discover interpretable algorithms. Moreover, the stopping criteria used in some of these algorithms are an interesting novel recombination of known principles in CGLS termination analysis.

[5] In finite precision, there might still be some error introduced by LSQR's running estimate of $\|\hat{A}\|$ through its Frobenius norm.

More details on the task from Section 5. The initial program on this task achieves $f_{\text{res}} = 1 + 5.8\times10^{-8}$, well below the target of $f_{\text{res}} \leq 1+10^{-4}$, so one might suspect that runtime improvements can come purely from stopping earlier. We show that the change in termination condition for the discovered algorithm only accounted for $1.12\times$ speedup, and most of the gain was due to moving away from sparse sign sketches in favor of random row sampling. See Figure 7 for a breakdown of each improvement and (Listing 3, Appendix H) for the full program.

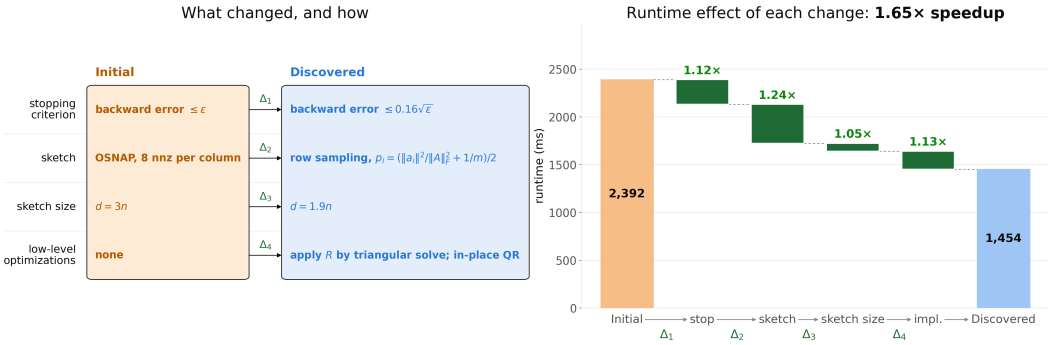

Figure 7: Speedup decomposition of the algorithm discovered on matrices of size $(30000, 2000)$, $\kappa = 10^{10}$, and $\varepsilon = 10^{-4}$. **Left.** Differences between the initial program (in orange) used in the discovery process and the best one (in blue). **Right.** Breakdown of performance gains from each change implemented.

Algorithm Analysis for size $(20000, 1000)$ and conditioning 10^6. The second task is the one with size $(20000, 1000)$, condition number 10^6 and accuracy target 10^{-4}, visualized in Figure 8. Here the discovered algorithm differs substantially from the initial program, switching from preconditioning to a direct Cholesky solver, which reduces runtime to roughly half. This shows that our framework can adapt to the task at hand and explore algorithms that are very different from the seed.

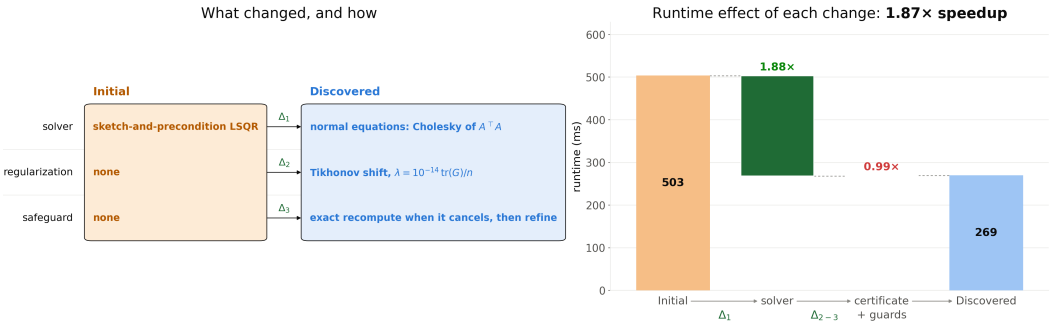

Figure 8: Speedup decomposition of the algorithm discovered on matrices of size $(20000, 1000)$, $\kappa = 10^6$, and $\varepsilon = 10^{-4}$. **Left.** Differences between the initial program (in orange) used in the discovery process and the best one (in blue). **Right.** Breakdown of performance gains from each change implemented.

Algorithm Analysis for size $(20000, 2000)$ and conditioning 10^6. In this task, most of the speedup came from changing the preconditioner from R^{-1} coming from the QR decomposition of the sketched data matrix to $L^{-\top}$ coming from the Cholesky decomposition of the sketched Gram matrix.

An interesting discovery in this task came from the termination condition. The accuracy target on Equation (2) is equivalent to

$$\|A(x - x^*)\|_2^2 \leq \left(1 - \frac{1}{(1+\varepsilon)^2}\right) \|Ax - b\|_2^2 = \eta \|Ax - b\|_2^2 \tag{5}$$

and the left-hand side can be written in terms of the Gram matrix and the gradient of the quadratic objective function $g = A^\top(Ax - b)$ as

$$\|A(x - x^*)\|_2^2 = g^\top (A^\top A)^{-1} g$$

Since the sketched matrix A_s contains a subset of the rows of A, we have $A_s^\top A_s \preceq A^\top A$, so we can upper bound f_{res} exactly through

$$g^\top (A_s^\top A_s)^{-1} g \leq 0.9\eta \|Ax - b\|_2^2$$

This uses only information available to the iterative algorithm, i.e. the sketched Gram matrix and the current residual. The 0.9 factor is added for slack. Although mathematically well-motivated, this stopping criterion does not provide a speedup on this task.

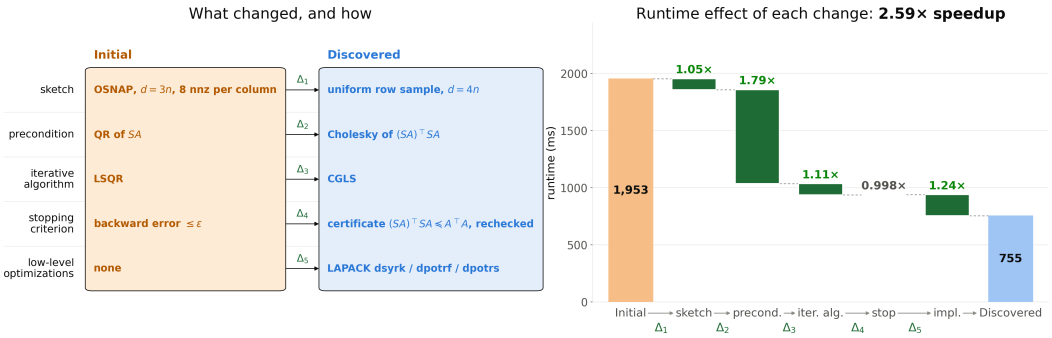

Figure 9: Speedup decomposition of the algorithm discovered on matrices of size $(20000, 2000)$, $\kappa = 10^6$, and $\varepsilon = 10^{-4}$. **Left.** Differences between the initial program (in orange) used in the discovery process and the best one (in blue). **Right.** Breakdown of performance gains from each change implemented.

Algorithm Analysis for size $(30000, 2000)$ **and conditioning** 10^6**.** The algorithm discovered for this task was also a randomized iterative solver, but with a 1-sparse sketch followed by a Cholesky-preconditioned CGLS. The termination condition is also non-standard, and we can explain it as follows. Let $r = Ax - b$, and $B = AR^{-1}$, where R is the Cholesky factor of $(SA)^\top(SA)$, i.e. the preconditioner. The discovered stopping criterion is

$$\|B^\top r\|_2 \leq \frac{\sqrt{\eta}}{1.1\|S\|_2} \|r\|_2 \tag{6}$$

where $\eta = \left(1 - \frac{1}{(1+\varepsilon)^2}\right)$. This is a provably sufficient stopping criterion for the accuracy target in exact arithmetic. Starting from the equivalent condition Equation (5) for $f_{\text{res}} \leq 1 + \varepsilon$, the right-hand side can be written as

$$\|A(x - x^*)\|_2^2 = (B^\top r)^\top (B^\top B)^{-1} (B^\top r) \leq \frac{\|B^\top r\|_2^2}{\sigma_{\min}(B)^2} \tag{7}$$

Note that the matrix SB has orthonormal columns, because $B = AR^{-1}$ where R is the Cholesky factor of $(SA)^\top(SA)$. Therefore, for any vector $y \in \mathbb{R}^n$,

$$\|y\|_2 = \|SBy\|_2 \leq \|S\|_2 \|By\|_2$$

This implies that $\sigma_{\min}(B) \geq \|S\|_2^{-1}$. Combining this with Equation (7) and Equation (5) shows that the stopping criterion implies $f_{\text{res}} \leq 1 + \varepsilon$. The algorithm builds a sketch matrix with (normalized) 1-sparse columns and rows with disjoint support, so they are orthonormal, hence $\|S\|_2 = 1$ by construction and does not need to be computed. This is what makes the bound Equation (6) efficiently computable by the iterative algorithm, since $\|r\|$ and $\|B^\top r\|$ are already available by CGLS iterates. Many of these ingredients are known in standard CGLS literature (Björck, 2024; Saad, 2003), and related spectral analysis of preconditioners appears in (Arioli & Duff, 2015). However, these ideas have not been made practical in standard scientific computing libraries, and, to the best of our knowledge, the application to f_{res} instead of backward error is novel. Contrary to the previous task, the new stopping criterion provided a $1.11\times$ runtime speedup relative to the default CGLS stopping rule (Figure 10).

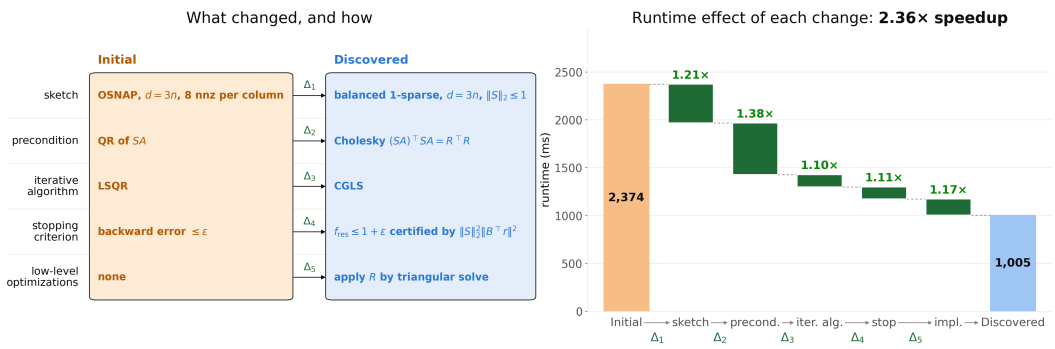

Figure 10: Speedup decomposition of the algorithm discovered on matrices of size $(30000, 2000)$, $\kappa = 10^6$, and $\varepsilon = 10^{-4}$. **Left.** Differences between the initial program (in orange) used in the discovery process and the best one (in blue). **Right.** Breakdown of performance gains from each change implemented.

G Other ideas explored

This appendix collects a couple of approaches that were considered for part of this project and later dropped from the final version of our framework, but they might be of interest to some readers.

G.1 Deterministic robust proxy for runtime

Since measured runtime is noisy and depends on many hardware and software-specific factors that are hard to control, we investigated using a deterministic proxy for runtime as score function for ShinkaEvolve. We took inspiration from the Roofline Model (Williams et al., 2009; Gholami et al., 2024) to measure the "speed" of each operation according to its arithmetic intensity. Given a machine with peak compute throughput P (i.e. number of FLOPs per second) and peak memory throughput W (i.e. number of bytes moved per second), the performance of an operation op under the roofline model is given by

$$P_{op} = \min \begin{cases} P \\ W \cdot I_{op} \end{cases}$$

where I_{op} is the arithmetic intensity, i.e. the ratio between the number of FLOPs performed and the number of bytes moved. Since the peak compute throughput is typically much larger than the peak memory throughput, having higher arithmetic intensity leads to higher performance. We calibrated the throughput of general matrix-matrix multiplication on our machine and defined the weight of an operation as

$$w(op) = \frac{P_{gemm}}{P_{op}}$$

We then used it to build a "wrapper" library around Numpy and Scipy containing common linear algebra operations used by least squares solvers, which also returned the FLOPs performed weighted by their throughput. For a program p composed of basic linear algebra operations op, its weighted cost is

$$C_p = \sum_{op \in p} \text{FLOPS}(op) w(op)$$

Rather than reproducing the precise runtime, the purpose of the proxy is to rank programs according to their runtime, so it can be used as score function during the discovery phase. This was especially promising for least squares algorithms, which use a composition of BLAS-2 and BLAS-3 operations. This succeeded in reducing the variance of runtime measurement, however it introduced too much bias which made it much less competitive against the final version of the score function we use. There are two main issues: (i) a single roofline is often not sufficient to model performance on modern machines with different memory levels and accelerators, and (ii) having to describe the library to the LLM prompt introduced a lot of bias in the proposed algorithms.

G.2 Minimizing residual error subject to a time budget

Our evaluation function is designed to minimize runtime subject to a target accuracy level. The reversed question is also interesting, i.e. finding the algorithm that maximizes accuracy subject to a time budget. We ran a few experiments with this score function and the algorithms we discovered were often inferior to the ones found by our current setup. One reason is that when the same accuracy can be reached by two algorithms that are both inside the time budget, this objective function does not discriminate between them and might choose the slower one.

H Discovered Practical Programs for Least Squares

In this section, we present the complete programs discovered by our framework for the tasks discussed in Appendix F. The full list of programs for all tasks considered in the paper will be released in a public repository at the time of publication.

Listing 3: Program discovered for the task of size $(30000, 2000)$, conditioning 10^{10}, and accuracy target 10^{-4}

```
# EVOLVE-BLOCK-START
import numpy as np
from scipy import linalg as sla
from scipy.linalg.lapack import dtrtrs
from scipy.sparse.linalg import LinearOperator, lsqr

_TINY2 = 1e-40

def solve(A, b, eps, iter_budget):
    """Solve min_x ||A x - b||_2 using sampled-QR preconditioned LSQR."""
    m, n = A.shape

    if n == 0:
        return np.empty(0, dtype=np.float64)

    # A modest reduction from 2n lowers both sampled-matrix construction and
    # QR cost while retaining a well-conditioned preconditioned operator.
    d = min(m, max(n + 16, int(1.9 * n)))

    if d == m:
        SA = np.array(A, dtype=np.float64, order="F", copy=True)
        Sb = np.array(b, dtype=np.float64, copy=True)
    else:
        # Reuse the energy array as the probability array to reduce allocations
        # and full memory passes.
        probabilities = np.einsum("ij,ij->i", A, A, optimize=True)
        total = float(probabilities.sum())

        if not np.isfinite(total) or total <= 0.0:
            return np.zeros(n, dtype=np.float64)

        probabilities *= 0.5 / total
        probabilities += 0.5 / m
        probabilities /= probabilities.sum()

        # Sampling without replacement avoids wasting sketch rows on duplicates.
        idx = np.random.choice(m, size=d, replace=False, p=probabilities)

        inclusion = d * probabilities[idx]
        np.minimum(inclusion, 1.0, out=inclusion)
        scale = 1.0 / np.sqrt(inclusion)

        SA = np.array(A[idx], dtype=np.float64, order="F", copy=True)
        SA *= scale[:, None]
        Sb = b[idx] * scale

    # Form this before QR overwrites the sampled matrix.
    rhs = SA.T @ Sb

    R = sla.qr(
        SA,
        mode="r",
        overwrite_a=True,
        check_finite=False
    )
    R = np.asfortranarray(R)

    # Warm start from the sampled normal equations.
    w, info = dtrtrs(R, rhs, lower=0, trans=1, overwrite_b=1)
    if info != 0:
        w = sla.solve_triangular(
            R, rhs, trans="T", lower=False, check_finite=False
        )

    x0, info = dtrtrs(R, w, lower=0, trans=0, overwrite_b=1)
```

```
    if info != 0:
        x0 = sla.solve_triangular(
            R, w, lower=False, check_finite=False
        )

    residual = b - A @ x0
    residual_norm2 = float(residual @ residual)
    if residual_norm2 <= _TINY2:
        return x0

    AT = A.T

    def matvec(v):
        z, _ = dtrtrs(R, v, lower=0, trans=0)
        return A @ z

    def rmatvec(u):
        g = AT @ u
        z, _ = dtrtrs(R, g, lower=0, trans=1, overwrite_b=1)
        return z

    operator = LinearOperator(
        (m, n),
        matvec=matvec,
        rmatvec=rmatvec,
        dtype=np.float64
    )

    # Objective error is quadratic in the normalized normal-equation residual
    # after sketch preconditioning. The cap also prevents an excessively loose
    # test when callers request low accuracy.
    target = max(float(eps), 0.0)
    atol = min(1.0e-2, 0.16 * np.sqrt(target))

    y = lsqr(
        operator,
        residual,
        atol=atol,
        btol=0.0,
        conlim=0.0,
        iter_lim=int(iter_budget),
        show=False
    )[0]

    correction, info = dtrtrs(
        R, y, lower=0, trans=0, overwrite_b=1
    )
    if info != 0:
        correction = sla.solve_triangular(
            R, y, lower=False, check_finite=False
        )

    return x0 + correction
# EVOLVE-BLOCK-END
# ...
```

Listing 4: Program discovered for the task of size $(20000, 1000)$, conditioning 10^6, and accuracy target 10^{-4}

```
# EVOLVE-BLOCK-START
import numpy as np
from scipy import linalg as sla

def solve(A, b, eps, iter_budget):
    """
    Super-fast overdetermined least-squares solver:
    1) Solves normal equations via fast Cholesky factorization.
    2) Estimates optimal residual and gradient in O(n^2) to perform
       the exact stopping check without doing costly O(mn) matvecs.
    3) Falls back to exact computation and one-step refinement if needed.
    """
    m, n = A.shape

    # 1. Compute AtA and Atb using optimized NumPy GEMM/GEMV
    AtA = A.T @ A
    Atb = A.T @ b

    # 2. Diagonal boost for stability
    tr = float(AtA.diagonal().sum())
    lam = max(1e-14 * tr / n, 1e-20)

    # Add the boost to AtA in-place (AtA is a temporary copy, so this is safe)
    np.fill_diagonal(AtA, AtA.diagonal() + lam)

    # 3. Cholesky factorization
    try:
        c, lower = sla.cho_factor(AtA, lower=False, overwrite_a=False, check_finite=False)
        x = sla.cho_solve((c, lower), Atb, overwrite_b=False, check_finite=False)
    except Exception:
        # Robust fallback for extreme cases
```

```
        return sla.lstsq(A, b, lapack_driver="gelsy", check_finite=False)[0]

    # 4. Ultra-fast exact stopping test
    b_norm2 = b.dot(b)
    if b_norm2 <= 1e-20:
        return np.zeros(n, dtype=A.dtype)

    tol_factor = (2.0 * eps + eps**2) / (1.0 + eps)**2

    # O(n^2) gradient and residual estimation
    AtA_x = AtA @ x
    unreg_AtA_x = AtA_x - lam * x
    g = Atb - unreg_AtA_x
    r_norm2 = b_norm2 - 2.0 * x.dot(Atb) + x.dot(unreg_AtA_x)

    # If the residual might have had catastrophic cancellation (highly consistent system),
    # fall back to exact O(mn) calculation.
    if r_norm2 < 1e-11 * b_norm2:
        r = b - A @ x
        r_norm2 = r.dot(r)
        g = A.T @ r

    dx = sla.cho_solve((c, lower), g, check_finite=False)
    w_norm2 = max(g.dot(dx), 0.0)

    # Typical case: returns immediately here after almost zero extra cost
    if w_norm2 <= tol_factor * r_norm2:
        return x

    # 5. Iterative Refinement
    x += dx

    # Small budget loop for refinement; usually 1 step is more than enough
    for _ in range(min(5, iter_budget)):
        r = b - A @ x
        r_norm2 = r.dot(r)
        if r_norm2 <= 1e-20:
            return x
        g = A.T @ r
        dx = sla.cho_solve((c, lower), g, check_finite=False)
        w_norm2 = max(g.dot(dx), 0.0)
        if w_norm2 <= tol_factor * r_norm2:
            return x
        x += dx

    return x
# EVOLVE-BLOCK-END
# ...
```

Listing 5: Program discovered for the task of size $(20000, 2000)$, conditioning 10^6, and accuracy target 10^{-4}

```
# EVOLVE-BLOCK-START
import numpy as np
from scipy import linalg as sla
from scipy.linalg.blas import dsyrk
from scipy.linalg.lapack import dpotrf, dpotrs, dposv

def solve(A, b, eps, iter_budget):
    """
    Row-subsampled Cholesky-preconditioned CG for dense least squares.

    For g = A.T @ (b - A@x), the excess squared residual is
        g.T @ inv(A.T@A) @ g.
    If G = As.T@As is formed from any subset of A's rows, then
        G <= A.T@A
    in the positive-semidefinite order, and consequently
        g.T @ inv(A.T@A) @ g <= g.T @ inv(G) @ g.
    This supplies a cheap, observable stopping certificate.
    """
    m, n = A.shape
    At = A.T
    requested = max(0, int(iter_budget))
    tolerance = float(eps)

    # Four rows per unknown generally gives a strong preconditioner while
    # requiring only 40% of the full Gram work when m/n is ten.
    sample_size = min(m, 4 * n)

    if tolerance > 0.0 and requested > 0 and sample_size < m:
        indices = np.random.choice(m, size=sample_size, replace=False)
        indices.sort()

        As = A[indices, :]
        bs = b[indices]
        G = dsyrk(1.0, As.T, trans=0, lower=0)
        sampled_rhs = As.T @ bs

        factor, factor_info = dpotrf(
            G, lower=0, clean=0, overwrite_a=True
```

```
        )

        if factor_info == 0:
            x, solve_info = dpotrs(
                factor, sampled_rhs, lower=0, overwrite_b=True
            )

            # Release the large sampled copy before repeatedly scanning A.
            del As, bs, G, sampled_rhs, indices

            if solve_info == 0 and np.all(np.isfinite(x)):
                residual = b - A @ x
                gradient = At @ residual

                z, z_info = dpotrs(
                    factor, gradient, lower=0, overwrite_b=False
                )
                rho = float(np.dot(gradient, z))
                residual_sq = float(np.dot(residual, residual))

                # Equivalent to 1 - 1/(1+eps)^2, evaluated without
                # cancellation for small eps.
                target_fraction = (
                    tolerance * (2.0 + tolerance)
                    / ((1.0 + tolerance) * (1.0 + tolerance))
                )
                certificate_limit = 0.90 * target_fraction

                if (
                    z_info == 0
                    and np.isfinite(rho)
                    and rho >= 0.0
                    and rho <= certificate_limit * residual_sq
                ):
                    return x

                if z_info == 0 and np.isfinite(rho) and rho > 0.0:
                    direction = z.copy()

                    # Bound the speculative work: an unusually poor sample
                    # is cheaper and safer to abandon for the direct fallback.
                    max_iterations = min(requested, 24)

                    for _ in range(max_iterations):
                        Adirection = A @ direction
                        Hdirection = At @ Adirection
                        denominator = float(np.dot(direction, Hdirection))

                        if (
                            not np.isfinite(denominator)
                            or denominator <= 0.0
                        ):
                            break

                        alpha = rho / denominator
                        x += alpha * direction
                        residual -= alpha * Adirection
                        gradient -= alpha * Hdirection

                        z_new, z_info = dpotrs(
                            factor, gradient, lower=0,
                            overwrite_b=False
                        )
                        if z_info != 0:
                            break

                        rho_new = float(np.dot(gradient, z_new))
                        if not np.isfinite(rho_new) or rho_new < 0.0:
                            break

                        residual_sq = float(np.dot(residual, residual))

                        if rho_new <= certificate_limit * residual_sq:
                            # Recompute both quantities from A to eliminate
                            # accumulated CG recurrence error before accepting.
                            true_residual = b - A @ x
                            true_gradient = At @ true_residual
                            true_z, true_info = dpotrs(
                                factor, true_gradient, lower=0,
                                overwrite_b=False
                            )

                            if true_info == 0:
                                true_certificate = float(
                                    np.dot(true_gradient, true_z)
                                )
                                true_residual_sq = float(
                                    np.dot(true_residual, true_residual)
                                )

                                if (
                                    np.isfinite(true_certificate)
```

```
                                    and true_certificate >= 0.0
                                    and true_certificate
                                    <= certificate_limit * true_residual_sq
                                ):
                                    return x

                                # Restart CG from the accurately recomputed
                                # gradient if the recurrence was optimistic.
                                if (
                                    np.isfinite(true_certificate)
                                    and true_certificate > 0.0
                                ):
                                    residual = true_residual
                                    gradient = true_gradient
                                    z_new = true_z
                                    rho_new = true_certificate
                                    direction = z_new.copy()
                                    rho = rho_new
                                    continue
                            break

                        if rho_new == 0.0:
                            return x

                        beta = rho_new / rho
                        direction *= beta
                        direction += z_new
                        rho = rho_new

    # Dependable fallback for singular samples, nearly consistent systems,
    # exhausted budgets, or unusually slow preconditioned convergence.
    full_gram = dsyrk(1.0, At, trans=0, lower=0)
    full_rhs = At @ b
    _, x, info = dposv(
        full_gram, full_rhs, lower=0,
        overwrite_a=True, overwrite_b=True
    )

    if info == 0:
        return x

    # Rank-revealing QR preserves the original unregularized objective.
    x, _, _, _ = sla.lstsq(
        A, b,
        cond=None,
        overwrite_a=False,
        overwrite_b=False,
        check_finite=False,
        lapack_driver="gelsy",
    )
    return x
# EVOLVE-BLOCK-END
# ...
```

Listing 6: Program discovered for the task of size $(30000, 2000)$, conditioning 10^6, and accuracy target 10^{-4}

```
# EVOLVE-BLOCK-START
import numpy as np
from scipy import linalg as sla
from scipy.sparse import csr_matrix

_TINY = 1e-20
_RNG = np.random.default_rng()

def _osnap(m, d):
    """Balanced signed one-sparse embedding with deterministic ||S||_2 <= 1."""
    cols = np.arange(m, dtype=np.int32)
    rows = cols % d
    _RNG.shuffle(rows)
    counts = np.bincount(rows, minlength=d)

    signs = _RNG.integers(0, 2, size=m, dtype=np.int8).astype(np.float64)
    signs *= 2.0
    signs -= 1.0
    signs /= np.sqrt(counts[rows])

    S = csr_matrix((signs, (rows, cols)), shape=(d, m))
    return S, 1.0

def solve(A, b, eps, iter_budget):
    """Solve min_x ||Ax-b|| with OSNAP-preconditioned, certified CGLS."""
    m, n = A.shape
    if n == 0:
        return np.empty(0, dtype=np.float64)

    # A 3n-row sketch provides a highly reliable preconditioner.
    d = min(m, 3 * n)
    S, snorm_bound = _osnap(m, d)
    SA = S @ A
```

```
    Sb = S @ b
    del S

    # Compute SA.T @ SA using highly-optimized BLAS GEMM.
    G = SA.T @ SA
    try:
        R = sla.cholesky(G, lower=False, overwrite_a=True, check_finite=False)
    except sla.LinAlgError:
        # Robust fallback if normal-equation rounding loses definiteness.
        R = np.linalg.qr(SA, mode="r")

    if not R.flags.f_contiguous:
        R = np.asfortranarray(R)

    # Cache LAPACK dispatch for the triangular solves outside the loop.
    trtrs = sla.get_lapack_funcs("trtrs", (R,))

    # Warm start: solve R.T @ R @ x = SA.T @ Sb
    rhs = SA.T.dot(Sb)
    z = trtrs(R, rhs, lower=0, trans=1, unitdiag=0, overwrite_b=1)[0]
    x = trtrs(R, z, lower=0, trans=0, unitdiag=0, overwrite_b=1)[0]

    del SA, Sb, G, rhs, z

    r = b - A.dot(x)
    rnorm_sq = float(np.dot(r, r))
    if rnorm_sq <= _TINY:
        return np.asarray(x, dtype=np.float64)

    # Certified stopping criterion based on the sketch norm.
    e = max(float(eps), 0.0)
    eta_sq = e * (2.0 + e) / ((1.0 + e) ** 2) if e > 0.0 else 0.0
    stop_ratio = eta_sq / (1.21 * snorm_bound * snorm_bound)

    # CGLS on B = A R^{-1}, updating x directly.
    g = A.T.dot(r)
    g_res = trtrs(R, g, lower=0, trans=1, unitdiag=0, overwrite_b=1)[0]
    gamma = float(np.dot(g_res, g_res))

    if gamma <= stop_ratio * rnorm_sq:
        return np.asarray(x, dtype=np.float64)

    p = g_res.copy()
    q = np.empty(m, dtype=np.float64)
    A_T_r = np.empty(n, dtype=np.float64)
    direction_x = np.empty(n, dtype=np.float64)

    budget = max(0, int(iter_budget))

    for _ in range(budget):
        np.copyto(direction_x, p)
        dir_res = trtrs(R, direction_x, lower=0, trans=0, unitdiag=0, overwrite_b=1)[0]

        A.dot(dir_res, out=q)
        qq = float(np.dot(q, q))

        if not np.isfinite(qq) or qq <= _TINY:
            break

        alpha = gamma / qq

        # Perform memory-saving, in-place scaling and additions
        dir_res *= alpha
        x += dir_res
        q *= alpha
        r -= q

        rnorm_sq = float(np.dot(r, r))
        if rnorm_sq <= _TINY:
            return np.asarray(x, dtype=np.float64)

        A.T.dot(r, out=A_T_r)
        at_res = trtrs(R, A_T_r, lower=0, trans=1, unitdiag=0, overwrite_b=1)[0]
        gamma_new = float(np.dot(at_res, at_res))

        if gamma_new <= stop_ratio * rnorm_sq:
            return np.asarray(x, dtype=np.float64)

        if not np.isfinite(gamma_new) or gamma <= _TINY:
            break

        p *= gamma_new / gamma
        p += at_res
        gamma = gamma_new

    return np.asarray(x, dtype=np.float64)
# EVOLVE-BLOCK-END
# ...
```