Precision-Aware Automated Discovery of Numerical Function Approximations via Landscape-Adaptive Search and LLM Agents

Toghrul Abbasov [*1] **Limeng ZHANG** [*2] **Zhihang Lin** [2] **Zhenyang Xiao** [3] **Xiaoxiao Tu** [2] **Zirong Zeng** [2] **Wentao Lin** [2] **Zheng Lingchao** [4] **Li Jun** [4] **Yuwei Fan** [4] **Tian Ding** [5] **Congliang Chen** [6 5] **Ruoyu Sun** [2 6 5]

Abstract

Elementary-function approximations are core components of numerical software, yet their design remains a manual, per-function, per-precision exercise. Reduced precision makes this substantially harder: designs that are accurate in real arithmetic can suffer catastrophic cancellation, intermediate overflow, or silent saturation when evaluated natively at the deployment precision. AutoNumerics-Zero recast this design step as evolutionary search over computation graphs, but its CMA-ES-based inner constant solver can discard good graph structures after poor constant optimization, and its outer workflow still requires manual decisions about interval partitioning and per-piece strategy selection. We address both limitations with a precision-aware, three-level pipeline: an *outer* language-model agent for partitioning, strategy selection, verification, and retry; a *middle* AutoNumerics-Zero-inspired evolutionary search over short arithmetic DAGs; and an *inner* fitter that combines Levenberg–Marquardt with Nelder–Mead on the requested rounded-arithmetic metric. On identical AutoNumerics-Zero 2^x candidates, the inner loop matches or improves published optima while avoiding large CMA-ES failures. End-to-end tanh experiments against same-precision OpenLibm-style baselines show that the full pipeline improves native-precision kernels: in fp16 tanh, it reduces max ULP from 287 to 2 on a dense validation grid.

[*]Equal contribution [1]School of Computing, Korea Advanced Institute of Science and Technology [2]The Chinese University of Hong Kong, Shenzhen [3]Peking University [4]Huawei Technologies Ltd. [5]Shenzhen Research Institute of Big Data [6]Shenzhen Loop Area Institute. Correspondence to: Tian Ding <dingtian@sribd.cn>, Congliang Chen <chencongliang@slai.edu.cn>, Ruoyu Sun <sunruoyu@cuhk.edu.cn>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

1. Introduction

Numerical approximations of elementary and special functions—`exp`, `log`, `sin`, `tanh`, `erf`, `lgamma`, and their relatives—are performance-critical components of the inner loops of transformer inference (Vaswani et al., 2017), ODE/PDE solvers (Hairer et al., 1993), signal-processing kernels (Oppenheim, 1999), and every standard `libm` (Muller, 2006). The dominant design methodology remains manual: a numerical analyst derives a minimax polynomial, a Padé approximant, or a staged range-reduction scheme in real arithmetic or fp64, and the result is compiled to the target precision for deployment. This separation is benign at fp32 and fp64, where the gap between design and deployment precision is dominated by a single final rounding, but breaks down at reduced precision: as machine-learning workloads move into fp16, bf16, the relative rounding per operation grows by orders of magnitude and the representable range narrows. Designs that are optimal in real arithmetic are therefore strictly suboptimal under the deployment format, and numerically unusable in non-trivial cases. The textbook identity $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$, for instance, exhibits catastrophic cancellation near the origin; although fp64 retains sufficient precision for the result to remain within a few ULP of the true value, fp16 amplifies this effect, leading to errors of tens of ULP—worse than a piecewise-linear baseline. Related failure modes include overflow of intermediate exponentials, loss of monotonicity under rounding, and silent saturation. Recovering low-precision accuracy therefore requires a *separate* design pass under the target floating-point model: cancelling forms must be rearranged, range reductions must be recomputed in the target format, and constants must be re-tuned against rounded-arithmetic error rather than a real-arithmetic surrogate. In current practice this pass is manual and specialist-driven.

Real et al. (2023), in AutoNumerics-Zero, recast the design step as a discrete search problem: a numerical implementation is represented as a directed acyclic graph (DAG) with real-valued constants, and a population of such DAGs is evolved under structured mutation and multi-objective selection on (error, op-count), with the constants inside each candidate tuned by a local optimizer. On several primitives

the system matches—and sometimes improves on—hand-crafted state of the art. Two limitations, however, prevent it from replacing the hand-engineering pipeline. First, the inner constant optimizer is the off-the-shelf CMA-ES (Hansen & Ostermeier, 2001). This zero-order sampler estimates curvature through sampling; near sharp minima, this introduces noise into its covariance estimates and causes unreliable convergence. Second, the outer workflow remains a human responsibility, and is both a throughput bottleneck and an error source.

This paper addresses both limitations with a unified, precision-aware framework. Our system, described in Section 3, is organized as *three nested loops*: an *outer* loop driven by a language-model agent that plans the workflow; a *middle* loop that evolves a population of DAGs under mutation and selection; and an *inner* loop that fits real-valued constants against the target floating-point model. The main contributions are in the inner and outer loops.

(i) A landscape-adaptive inner optimizer. We replace the CMA-ES constant solver with a two-stage solver matched to the geometry of the constant-fitting problem: Levenberg–Marquardt exploits Jacobian-based curvature information to navigate the narrow, anisotropic basin efficiently, and Nelder–Mead then refines the solution on the piecewise-constant staircase of the precision-exact metric where no gradient signal is available. The stages are motivated by an empirical landscape analysis and, on identical AutoNumerics-Zero DAG structures, the new solver matches or beats the published fp64 optima where CMA-ES exhibits errors several orders of magnitude above the optimum.

(ii) An LLM agent for outer-loop orchestration. We introduce an LLM agent that plans the outer workflow over a typed tool interface, including interval partitioning, per-piece strategy selection, verification, and retry. The agent consumes structured observations from search, baseline, and verifier tools and updates the approximation specification passed to the underlying function-approximation engine, without performing numerical evaluation itself.

(iii) End-to-end gains under explicit precision models. Across fp16 sigmoid, tanh, and sinh, and a local fp32 tanh searched core, the integrated pipeline finds compact piecewise kernels that improve target-precision accuracy over matching OpenLibm-style baselines. The largest gains occur in reduced precision: for example, pure-fp16 tanh reduces max ULP from 287 to 2 on a dense validation grid.

2. Related Work

Automated discovery of elementary-function approximations. The closest prior work is AutoNumerics-Zero (Real et al., 2023), which casts approximation design as evolutionary search over computation graphs using structured mutation and multi-objective selection on the error/op-count Pareto frontier. Classical automation produces optimal polynomial or rational approximations under analytic criteria: Sollya (Chevillard et al., 2010) embodies Remez-style algorithms and efficient L_∞ fitting (Brisebarre & Chevillard, 2007); Metalibm (Kupriianova & Lauter, 2014) compiles such approximations into deployable kernels; Muller (2006) surveys textbook range-reduction algorithms; `Herbie` (Panchekha et al., 2015) rewrites existing expressions to minimize floating-point error. Our outer agent invokes Sollya-style baselines for pieces where classical methods suffice and falls back to evolutionary search otherwise.

Correctly rounded and low-precision math libraries. RLIBM (Lim et al., 2021) formulates correctly-rounded approximation as a linear program over intervals of acceptable outputs, producing polynomial implementations provably correctly rounded for every input in formats such as bfloat16 and posit; CR-LIBM (Daramy et al., 2003) provides correctly rounded fp64 elementary functions via two-step evaluations. These systems are precision-aware in the same spirit as ours, but are restricted to polynomial approximants under hand-crafted range reductions; our search discovers the computation graph itself, including non-polynomial structure, with verification against the target floating-point model.

LLM agents and tool use. Tool-using LLMs (ReAct (Yao et al., 2023), Toolformer (Schick et al., 2023)) have catalyzed agent systems for mathematical and programming tasks. In mathematics, AlphaGeometry (Trinh et al., 2024) combines a neural language model with a symbolic deduction engine for olympiad geometry, illustrating the value of using language models to guide structured mathematical tools rather than replace them. Most directly related is FunSearch (Romera-Paredes et al., 2024), which uses an LLM as a program-mutation operator inside an evaluation-based selection loop.

3. System Design

The end-to-end flow, illustrated in Figure 1, proceeds in five stages. (1) The agent parses the natural-language request into a structured tuple (f, I, P), where f is the target function, I is the operating interval, and P is the precision model (input, compute, and output formats; rounding mode; error metric; threshold). (2) It invokes domain-analysis tools to detect periodicity, parity, and singularities of f on I, and partitions I into pieces $\{I_j\}$ whose boundaries respect the detected structure. (3) For each piece, the agent chooses a *strategy*: evolutionary graph search when the piece is structurally non-trivial, or a classical baseline (Chebyshev, minimax, or Taylor) when a closed-form approximant is already likely to be competitive. (4) Every piece routed to search enters the middle loop, which evolves a population of computation graphs; every candidate graph enters the

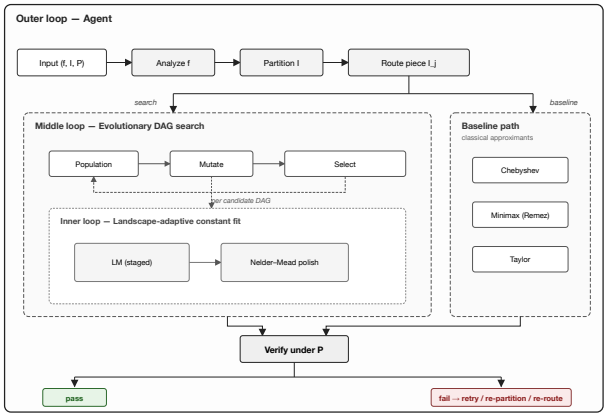


Figure 1. Three-loop pipeline. The *outer* loop, driven by a language-model agent, parses the request (f, I, P), analyzes f, partitions I, and routes each piece to either structured graph search or a classical baseline. Pieces routed to search enter the *middle* loop, an evolutionary search over DAGs adopted from AutoNumerics-Zero. Each candidate DAG enters the *inner* loop, which fits its real-valued constants under the precision model P. Verified pieces are stitched into a piecewise code artifact; failing pieces trigger agent-driven retry.

inner loop, which fits its constants against P. (5) The agent verifies each returned candidate against the precision model, re-plans failing pieces (by tightening sampling, subdividing, or switching strategy), and finally emits a single piecewise implementation when every piece has cleared the threshold. The three loops are composed so that each sees only the abstractions relevant to its layer: the agent sees verified piecewise candidates; the graph search sees structural proposals with their post-fit error; the inner optimizer sees a fixed DAG and a constant-fitting objective. The remainder of this section describes each loop in turn.

3.1. Outer Loop: Agent

The outer loop turns a natural-language request into a verified piecewise implementation. Its job is not to run any numerical primitive but to *decide what each piece of the function needs*: how to cut the interval, what method to apply, when a result is acceptable, and what to do when it is not. Formally, the outer loop maintains an observation $o_t = (S_t, B_t, V_t, H_t)$, where S_t is the current piecewise specification, B_t is the baseline profile, V_t contains verifier metrics and counterexamples, and H_t summarizes search health. The agent maps this observation to typed actions such as REVISE-SPEC, SUBMIT-SEARCH, STOP-SEARCH, VERIFY, and ACCEPT-PIECE. Here REVISE-SPEC covers both partition changes and numerical-search configuration changes, including sampling, metric thresholds, DAG-search budgets, mutation settings, optimizer settings, and per-piece strategy. Table 1 summarizes the tool families exposed to the agent. These names denote typed, sandboxed tools or scripts behind the interface; the LLM does not perform numerical evaluation itself, but chooses which tool to call and rewrites the specification from the returned structured observations. Some tools, such as MATHINSPECTOR, may use fixed-seed randomized or adaptive probing, but execution is resource-bounded, restricted to an allow-listed numerical library set, and logged with seeds when replayability is required.

Agent control interface. At each decision point the agent receives an observation $o_t = (S_t, B_t, V_t, H_t)$: the current APPROXSPEC and piece states S_t, baseline profile B_t, verifier output V_t including counterexamples, and search-health summary H_t containing best error, throughput, elapsed time, and stagnation counts. The action space is small and typed: WRITE-SPEC, UPDATE-SPEC, SUBMIT-SEARCH, STOP-RUN, VERIFY, SPLIT-PIECE, ROUTE-TO-BASELINE, and EMIT-CODE. Tool execution produces the next observation; the LLM is responsible only for selecting the next typed action and its arguments. This design makes the outer loop auditable: every numerical claim in the final report is produced by a verifier or profiler tool, not by free-form language generation.

Specification and decomposition. The agent first parses the request into the structured search problem (f, I, P)—target function, operating interval, and precision model (input/compute/output formats, rounding mode, error metric, acceptance threshold)—and fixes the sampling strategy, search budget, and stopping criteria that all downstream loops share. It then probes f on I for periodicity, parity, singularities, and monotonicity, partitions I into pieces $\{I_j\}$ whose boundaries respect that structure, and routes each piece: structurally non-trivial pieces enter the middle loop, while pieces for which a classical approximant (Chebyshev/minimax/Taylor) is already likely to clear the threshold are sent down a baseline path. This step controls how much compute the system spends and where.

Search, verification, and retry. For each piece, the agent can run DAG search or try classical approximation methods such as Chebyshev, minimax, or Taylor fitting. Candidates are verified under the requested precision model, producing error metrics and counterexamples. If a piece fails or stagnates, the agent revises the corresponding specification, such as its interval, sampling, metric, search budget, optimizer settings, or strategy, and reruns only that piece. Passing pieces are kept fixed.

3.2. Middle Loop: Evolutionary DAG Search

Each piece I_j routed to search invokes an independent run of the middle loop. We adopt the AutoNumerics-Zero formulation essentially unchanged at this layer; our role is restricted to exposing the configuration to the outer agent and consuming the returned candidates. We summarize the formulation for completeness.

Table 1. Outer-loop tool families. Numerical measurements are produced by typed tools; the LLM-mediated part is the decision of which tool to call next and how to update the piecewise specification from structured observations.

Tool family	Input	Output	Control type
MathInspector	Function, interval, precision model	Parity, monotonicity, singularity, saturation, and risk-region cues	Sandboxed scripted probe; fixed-seed or budgeted randomized/adaptive sampling
SpecCompiler	Agent plan, pieces, metrics, precision settings	Validated ApproxSpec or schema-level errors	Deterministic validator
SearchOrchestrator	Piece spec, worker budget, operation budget	Submitted, stopped, or resumed DAG-search jobs	Deterministic run manager; agent selects action
ProgressMonitor	Active task identifiers and search logs	Best error, throughput, stagnation count, candidate pointer	Deterministic poller
BaselineProfiler	Function, interval, baseline precision policy	Same-precision baseline errors and weak sub-intervals	Deterministic evaluator
PrecisionVerifier	Candidate DAG or generated code, reference oracle, grid	Max/mean ULP, max/mean absolute error, counterexamples	Deterministic rounded-arithmetic verifier
CodeEmitter	Verified per-piece candidates and dispatch boundaries	Piecewise C/Python artifact with dispatch logic	Deterministic code generator

DAG representation. A candidate implementation is a directed acyclic graph with four kinds of nodes: input nodes (the variables of f), constant nodes (real-valued parameters), operation nodes (one of $+, -, \times, \div$), and a single output node. A topological sort determines the evaluation order for code generation. After every mutation, redundancy elimination (constant folding, removal of non-output-reachable subgraphs) and cut-point/cut-edge analysis are applied to enforce graph validity and a canonical form.

Mutation. At every middle-loop step, the current population produces a larger offspring pool. Each offspring is sampled from one of four mutation types—*identity*, *edge rewiring*, *node deletion*, and *node insertion*—with weights chosen so that conservative mutations dominate and disruptive structural changes are rarer. The three non-trivial mutations each preserve input-to-output reachability via the cut-point analysis before committing the change.

Selection. Offspring and survivors are ranked by non-dominated sorting on the two objectives: max error under P from the inner-loop fit, and operation count. Diversity is preserved by retaining at most one DAG per (front, op-count) cell, with a distance-based tiebreak within each front. The top-ranked DAGs proceed to the next generation.

Asynchronous parallelism. Each search run is managed by a *coordinator* process that maintains the population and selection state. The coordinator dispatches candidate DAGs to a configurable pool of *worker* processes; each worker independently fits the constants of one DAG via the inner loop and returns the post-fit error. The worker count is set per-experiment by the agent (or defaults to the host CPU core count) and is allocated across concurrent pieces by the JobDispatcher so that multiple pieces can search in parallel without over-subscribing the machine.

The layer's role in our system is a component, not a contribution; what is new is that its configuration—operating interval, budget, metric, precision model—is authored by the outer agent rather than a human, and that its returned candidates are consumed by a verification loop that is itself agent-driven.

3.3. Inner Loop: Landscape-Adaptive Constant Fit

Given a DAG structure G with d constant slots $c \in \mathbb{R}^d$, the inner loop solves

$$c^\star \;=\; \arg\min_{c\in\mathbb{R}^d} \mathcal{E}_P\big(G(\cdot;\, c),\; f,\; I_j\big), \tag{1}$$

where $\mathcal{E}_P$ is the target error metric (max absolute, relative, or ULP error) evaluated under the precision model P through rounded floating-point execution of G.

Landscape properties. The constant-fitting objective $\mathcal{E}_P(c) = \max_{x\in X_{\text{test}}} |(G(x;c) - f(x))/(f(x)+\epsilon)|$ exhibits two properties that jointly govern solver selection. At large scale, the loss surface is highly non-convex with a *sharp* global minimum: profiling $\mathcal{E}_P$ along two dimensions in c-space reveals multiple basins and a global minimum at the bottom of a narrow, high-curvature valley, where the loss drops by several orders of magnitude over a region whose width is small relative to any reasonable initialization box. Figure 2a illustrates this behaviour for a representative DAG. At small scale, once a solver enters the correct basin, the precision-exact metric reduces to a piecewise-constant staircase with non-differentiable boundaries; Figure 2b shows a representative zoom into this structure.

Why standard solvers are insufficient. These two properties make it difficult for a single solver to perform well across scales. CMA-ES can in principle adapt its covariance C to align with a narrow valley, but doing so requires many generations of successful steps, making the evaluation cost prohibitive when each DAG evaluation under rounded arithmetic is expensive. Plain gradient descent is effectively stalled in such ill-conditioned valleys, with updates dominated by oscillations across high-curvature directions rather than progress along the valley floor. Quasi-Newton methods

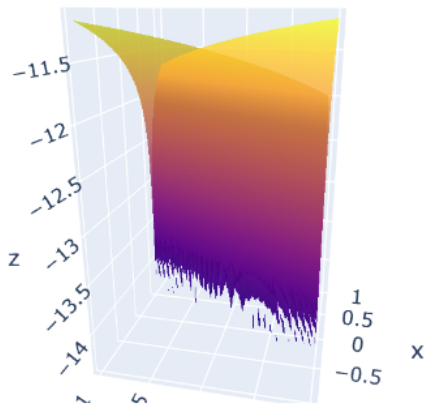
such as BFGS face similar difficulties, as their secant-based Hessian approximation requires many iterations to align with the highly anisotropic geometry of the valley. At the fine scale, gradient-based methods become ineffective due to the non-differentiable staircase structure. These observations motivate a two-stage procedure: a Levenberg–Marquardt stage (Marquardt, 1963) that exploits Jacobian-based curvature information to navigate the anisotropic basin efficiently, followed by a Nelder–Mead stage (Nelder & Mead, 1965) that refines the solution on the piecewise-constant surface without requiring gradients.

First stage: Levenberg–Marquardt. The residual vector $\mathbf{r}(\mathbf{c}) = \mathbf{y} - G(\mathbf{x}, \mathbf{c})$ makes the Gauss–Newton matrix $\mathbf{J}^\top\mathbf{J}$ available without forming a full Hessian. LM solves the damped normal equations

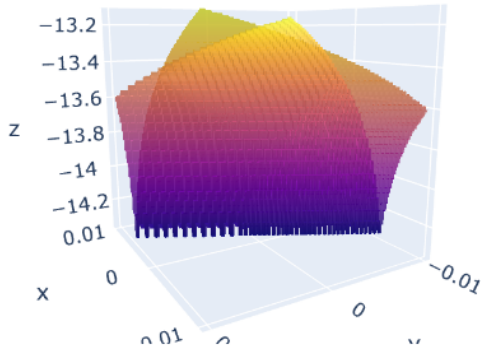
$$\left[\mathbf{J}^\top\mathbf{J} + \lambda\mathbf{I}\right]\boldsymbol{\delta} \;=\; \mathbf{J}^\top\mathbf{r}(\mathbf{c}), \tag{2}$$

with $\mathbf{J} = \partial G/\partial\mathbf{c}$ assembled by reverse-mode differentiation. The damping λ interpolates between Gauss–Newton ($\lambda \to 0$, fast local convergence) and gradient descent ($\lambda \to \infty$, robust steps from poor initializations). LM is particularly effective in this setting, as the Gauss–Newton approximation $\mathbf{J}^\top\mathbf{J}$ provides a problem-specific curvature estimate at each iteration without requiring it to be built up through secant updates or stochastic sampling, enabling rapid convergence within the narrow, high-curvature valley. Global non-convexity is handled by random restarts wrapping LM (Algorithm 1) rather than by the local solver.

Second stage: Nelder–Mead. Once LM has located the correct basin, the precision-exact metric $\mathcal{E}_P$ reduces to a piecewise-constant staircase with non-differentiable boundaries, making gradient-based refinement ineffective. Nelder–Mead addresses this directly: it maintains a simplex of $n+1$ vertices and updates the worst vertex by reflection, expansion, contraction, or shrinkage based purely on objective comparisons. Because the simplex moves by comparison rather than by gradient, it crosses plateau boundaries cleanly and settles on the correct plateau of $\mathcal{E}_P$ even where no descent direction is locally defined. LM navigates the anisotropic basin efficiently; Nelder–Mead then pushes the last 1–2 ULP onto the optimal plateau of the precision-exact metric. Algorithm 1 gives the end-to-end procedure.



(a) Large-scale: highly non-convex, with the global minimum at the bottom of a narrow, high-curvature valley.



(b) Zoom-in: the precision-exact metric is a piecewise-constant staircase with non-differentiable boundaries.

Figure 2. Loss-surface profiles of $\mathcal{E}_P$ along representative slices in c-space.

4. Experiments

4.1. Protocol

Baselines. The end-to-end baseline is derived from OpenLibm under the same precision setting as the candidate. For fp32, the baseline is the OpenLibm `float` path evaluated with fp32 inputs, fp32 local variables, fp32 intermediate values, and fp32 output. For fp16, OpenLibm does not define an official fp16 elementary-function API; our fp16 baseline is therefore a *pure-fp16 lowering* of the corresponding OpenLibm-style implementation: the same source-level formula/control-flow family is used, but inputs, constants after rounding, local variables, intermediate values, and outputs are all rounded to fp16. This baseline answers the same question as our candidates: what happens when the kernel is executed natively at the deployment precision? We do not compare fp16 candidates against fp32 or fp64 intermediate implementations, because that would measure a different kernel. For the inner-loop ablation (Section 4.2), the baseline is the CMA-ES constant solver used by AutoNumerics-Zero, run on the same DAG structures.

Metrics. Every reported row uses the same four metrics: MAX_ULP, MEAN_ULP, MAX_ABS, and MEAN_ABS. Ratios are always $\text{ratio} = \text{error}_{\text{DAG}}/\text{error}_{\text{baseline}}$, so values below one indicate improvement; ties are reported explicitly.

Algorithm 1 Two-stage constant fit for a DAG G.

Input: DAG G with d constants; precision-exact metric $\mathcal{E}_P$; stages T; perturbations-per-stage K; Gaussian perturbation scale σ; Nelder–Mead tolerance τ and budget B.
Output: fitted constants $c^\star$ and observed error $\mathcal{E}_P(G(\cdot; c^\star), f, I_j)$.
Stage 1: staged Levenberg–Marquardt
Draw an initial point c_{best} from a log-uniform prior over typical evolved-DAG constant magnitudes.
for $t = 1, 2 \ldots, T$ **do**
  Sample K Gaussian perturbations $\{c_{\text{best}} + \sigma\xi_k\}_{k=1}^{K}$ with $\xi_k \sim \mathcal{N}(0, I)$.
  $c_{\text{best}} \leftarrow \arg\min_k \mathcal{E}_P\big(\text{LM}(c_{\text{best}} + \sigma\xi_k)\big)$
end for
Stage 2: Nelder–Mead
$c^\star \leftarrow \textsc{NelderMead}(c_{\text{best}}, \mathcal{E}_P; \tau, B)$
Terminate on simplex diameter $< \tau$ *or budget* B *exhausted.*
return $c^\star$ and $\mathcal{E}_P(G(\cdot; c^\star), f, I_j)$

Evaluation sets. Sigmoid fp16 is evaluated on all positive fp16 inputs in $(0, 8]$ (18,432 points). Tanh fp16 is evaluated on a dense 40,000-point grid over $[-5, 5]$. Sinh fp16 is evaluated on all fp16 inputs in $[0, 11.75]$ covered by the exported positive-domain pieces (18,914 points). Tanh fp32 is evaluated over the searched core $[0, 0.7]$ by dense 200k-point fp32 grids on each accepted piece, $[0, 0.3]$, $[0.3, 0.5]$, and $[0.5, 0.7]$. For fp32 tanh, max metrics are the worst value over the three pieces and mean metrics are averaged across the three equal-size grids.

4.2. Inner-Loop Ablation: LM+Nelder–Mead vs. CMA-ES

We isolate the inner-loop contribution by re-fitting the constants of three DAG structures from AutoNumerics-Zero (f_6, f_8, f_{10}, all targeting 2^x) under four solvers: CMA-ES (the original inner loop), CMA-ES followed by LM, LM alone, and our two-stage LM + Nelder–Mead. The DAG structures and the published per-DAG fp64 error optima follow AutoNumerics-Zero; solver-reported MRE is evaluated in fp64. Results are summarized in Table 2.

Table 2. Inner-loop ablation on AutoNumerics-Zero 2^x DAGs. MRE is evaluated in fp64. **Bold** marks the best solver per target.

Method	f_6	f_8	f_{10}
Pub. opt. (fp64)	3.98×10^{-7}	1.32×10^{-11}	5.40×10^{-15}
CMA-ES	$\mathbf{2.80\times10^{-7}}$	1.39×10^{-8}	2.42×10^{-9}
CMA-ES + LM	6.25×10^{-7}	2.07×10^{-11}	7.10×10^{-15}
LM	6.25×10^{-7}	2.08×10^{-11}	5.44×10^{-15}
LM + Nelder–Mead	$\mathbf{2.80\times10^{-7}}$	$\mathbf{9.56\times10^{-12}}$	$\mathbf{4.21\times10^{-15}}$

Three findings stand out. (i) On f_6, whose landscape is well behaved, CMA-ES already improves on the published value and our solver matches that result; the inner-loop choice does not matter here. (ii) On f_8 and f_{10}, CMA-ES alone fails by 3–6 orders of magnitude (1.3×10^{-8} vs. a published 1.3×10^{-11} on f_8; 2.4×10^{-9} vs. 5.4×10^{-15} on f_{10}), consistent with the landscape analysis in Section 3.3: CMA-ES fails to reach the optimum on these ill-conditioned landscapes, leaving a gap of several orders of magnitude. (iii) Adding LM behind CMA-ES recovers the published optimum but does *not* surpass it, while LM + Nelder–Mead matches CMA-ES on f_6 and *beats* the published fp64 optimum on f_8 ($1.4\times$ lower MRE) and f_{10} ($1.3\times$ lower). The Nelder–Mead polish is the component that crosses the piecewise-constant plateau boundaries that LM alone cannot resolve.

4.3. Agent Partition Decisions

The outer loop matters because the hard decision is often not a new arithmetic primitive, but when to stop spending compute on the current specification and how to rewrite that specification.

Table 3 isolates the value of the outer agent. With the same 128-worker budget, a fixed-global search over $[0, 0.7]$ created 582,512 DAGs in 21 hours but plateaued at 1.19×10^{-7}, still $1.39\times$ worse than OpenLibm. The agent-created three-piece specification used 242,217 created DAGs across $s_0/s_1/s_2$ and accepted all three intervals. Notably, the fixed global run stopped at the same 1.19×10^{-7} plateau that the agent later observed on its hardest piece, $s_2 = [0.5, 0.7]$; after reallocation, that piece improved to 5.96×10^{-8}. The outer loop therefore contributes by changing the search problem exposed to the DAG engine, not by spending more created-DAG budget.

Controller ablation: is the LLM agent necessary? Table 3 compares only two controllers on a single function. To ask directly how much of the outer-loop benefit requires a language model, Table 4 broadens the comparison to the full controller spectrum on two native-fp16 functions, holding the middle DAG search and the inner LM+Nelder–Mead fitter *fixed* and giving every controller the same 90-worker, $\sim$6h budget. Only the outer policy changes: *fixed-global* searches the whole positive half as one piece; *fixed-uniform* cuts it into k=4 equal pieces; *rule-based* starts from one piece and subdivides deterministically at the worst verifier counterexample whenever a piece stagnates, using *no language model*; and *LLM agent* is the agent's own partition for that function, replayed under the common budget. Created DAGs is an outcome, not a control, and is reported per controller.

Three observations isolate the outer-loop contribution. (i) *Partitioning is decisive*: moving from a single global piece

Table 3. Fixed-global ablation for fp32 tanh on $[0, 0.7]$ under the same 128-worker budget. The fixed-global run searches the whole interval as one piece; the agent rewrites it into three focused pieces.

Controller	Fixed global	Agent outer loop
Domain decomposition	$[0, 0.7]$	$[0, 0.3]$, $[0.3, 0.5]$, $[0.5, 0.7]$
Worker budget	128	128
Created DAGs	582,512	242,217
Best-of DAGs	89,234	237,964
Best max abs	1.19×10^{-7}	2.98×10^{-8} / 2.98×10^{-8} / 5.96×10^{-8}
vs. OpenLibm	$1.39\times$ worse	$0.76\times$ / $0.50\times$ / $0.69\times$
Accepted pieces	0	3

Table 4. Controller ablation under a fixed inner/middle loop and a common 90-worker, ∼6h budget per controller. Only the outer partition/split policy changes; MAX ULP is the candidate's worst case on the same dense fp16 grids as Table 6, and created DAGs is an outcome, not a control. The rule-based controller uses no language model. The reduced shared budget makes absolute ULPs slightly higher than the converged kernels of Table 6 (e.g. sinh); the object of study is the comparison *across controllers at equal budget*.

	tanh fp16		sinh fp16	
Controller	DAGs	max ULP	DAGs	max ULP
Fixed-global	9,454	8	1,328	71
Fixed-uniform (k=4)	6,013	6	1,530	43
Rule-based (no LLM)	4,236	5	2,263	21
LLM agent	7,612	**2**	1,497	**10**

to any non-trivial partition cuts worst-case error sharply ($8 \to 2$ on tanh, $71 \to 10$ on sinh). (ii) *A non-LLM adaptive controller captures part of the gain, but not all of it*: the counterexample-driven rule-based controller improves on fixed-global ($8 \to 5$, $71 \to 21$) yet trails the agent, and it does so while *over-subdividing*—it splits to the piece-count cap (12 pieces) and thereby starves each piece of search budget (the fewest created DAGs per piece of any controller). Naive rules localize error but mis-budget it. (iii) *The LLM agent attains the lowest error on both functions* (2 and 10) with markedly fewer, better-placed pieces (3 and 8), reproducing the converged tanh-fp16 result of Table 6 under the controlled budget. The outer-loop benefit is therefore not merely "adaptive splitting helps"—the rule-based controller already demonstrates that—but that the agent decides *where* and *how much* to cut more judiciously than a fixed or rule-based policy at equal compute. The live agent layers further non-partition decisions (objective repair, symmetry reuse) on top of this partition, as the sigmoid case study below illustrates.

Sigmoid fp16 provides a case study of the outer agent. The agent did not simply launch a fixed search: it used symmetry to reduce the searched domain, split the positive half non-uniformly after stagnation signals, and changed the objective when the max-error metric stopped distinguishing candidates with different numbers of bad points. These decisions produced a six-piece positive kernel whose dense positive-half evaluation matches the baseline max ULP and reduces mean ULP and mean absolute error by $3.6\times$.

4.4. End-to-End Accuracy Results

Table 6 combines the same-precision baseline comparison with the relevant piecewise implementation information. Each row is a dense validation domain, and each metric cell is reported as baseline → DAG; lower is better, bold marks an improvement, and = marks a tie. This makes the fp16 worst-case gains visible while also showing the fp32 tanh tradeoff directly: over the searched core $[0, 0.7]$, the DAG improves mean ULP and mean absolute error, but it is not a worst-case or global `tanhf` replacement claim.

Expression size. The search objective includes expression size, so we also audit whether the selected DAGs are actually smaller than the same-precision OpenLibm-style baselines. Table 7 reports the comparison per reported interval rather than as a task-level average. DAG size is the arithmetic-node count exported by the search artifacts. Baseline size uses a source-expanded arithmetic count: scalar additions, subtractions, multiplications, and divisions are counted, while comparisons, branches, bit manipulation, `fabs`, sign routing, and constant loads are not. Crucially, `expf` and `expm1f` are expanded to their OpenLibm polynomial/range-reduction kernels instead of being counted as one opaque library call. Under this count, every reported DAG piece is smaller than the corresponding expanded baseline; even the most conservative grouped comparison is at least $1.3\times$ smaller, and most rows are closer to $2\times$ or more. This is an expression-level complexity comparison, not a direct runtime claim.

For sigmoid fp16, the agent searches only the positive half and uses $\sigma(-x) = 1 - \sigma(x)$ for the negative half. The OpenLibm-style pure-fp16 sigmoid already reaches max ULP 1, so there is no room to improve the worst case without exact rounding; the gain is in average accuracy, where mean ULP and mean absolute error each drop by $3.6\times$. For tanh fp16, the agent uses odd symmetry and three positive pieces. The strongest effect is tail-risk removal:

Table 5. Agent decisions in the sigmoid fp16 run. The agent improves the search problem before launching more computation: it reduces the domain, separates numerical regimes, and changes the objective when max error stops providing useful ranking signal.

Decision	Diagnostic signal	Agent action	Effect
Symmetry reduction	Sigmoid satisfies $\sigma(-x) = 1 - \sigma(x)$.	Search only $[0, 8]$; route the negative half by symmetry.	Cuts the searched domain in half with no extra DAG search.
Regime-aware splitting	Single-piece $[0, 8]$ stagnates at 1.46×10^{-3} after 17.2h, above the 5×10^{-4} target.	Split by numerical regime: $[0, .5]$, $[.5, 1]$, $[1, 2]$, $[2, 3]$, $[3, 4]$, $[4, 8]$.	Allocates more pieces near the high-density, high-curvature region and fewer pieces in saturation.
Objective repair	Max absolute error saturates near one fp16 ULP and no longer distinguishes one bad point from many bad points.	Switch search objective from max absolute error to mean absolute error; keep final verifier on max/mean ULP and absolute error.	On $[0, .5]$, nonzero-ULP points drop from 1417 to 137.

Table 6. Same-precision dense-validation results. Each metric cell shows baseline on the first line and our DAG on the second line; lower is better. Bold marks DAG improvements and = marks ties. The fp16 baseline is an OpenLibm-style implementation mechanically lowered so that variables, constants, and intermediates are fp16; the fp32 baseline is OpenLibm fp32.

Task	Domain	Kernel / role	max ULP	mean ULP	max abs	mean abs
sigmoid fp16	$(0, 8]$	6 positive pieces: $[0, .5]$, $[.5, 1]$, $[1, 2]$, $[2, 3]$, $[3, 4]$, $[4, 8]$	1	0.091	7.0×10^{-4}	4.46×10^{-5}
			1 =	**0.025**	$\mathbf{5.0\times10^{-4}}$	$\mathbf{1.23\times10^{-5}}$
tanh fp16	$[-5, 5]$	3 positive pieces plus odd symmetry: $[0, .25]$, $[.25, 1]$, $[1, 5]$	287	0.501	9.77×10^{-4}	1.61×10^{-4}
			2	**0.348**	9.77×10^{-4} =	$\mathbf{1.56\times10^{-4}}$
sinh fp16	$[0, 11.75]$	8 positive pieces with high-range refinement: $[0, 3]$, $[3, 5]$, $[5, 7]$, $[7, 9]$, $[9, 9.907]$, $[9.907, 10.375]$, $[10.375, 11]$, $[11, 11.75]$	1021	2.798	∞	∞
			7	**0.208**	$\mathbf{1.75\times10^{2}}$	$\mathbf{5.01\times10^{-1}}$
tanh fp32*	$[0, 0.7]$	searched core; 3 pieces: $[0, 0.3]$, $[0.3, 0.5]$, $[0.5, 0.7]$	2	0.459	8.58×10^{-8}	1.64×10^{-8}
			3	**0.399**	1.02×10^{-7}	$\mathbf{1.44\times10^{-8}}$
tanh fp32 local†	$[0.3, 0.5]$	central searched piece; dense 200k fp32 grid	2	0.511	6.01×10^{-8}	1.67×10^{-8}
			2 =	**0.415**	$\mathbf{4.61\times10^{-8}}$	$\mathbf{1.39\times10^{-8}}$

For sinh fp16, the OpenLibm-style pure-fp16 baseline overflows in an intermediate high-range `exp` path, so its absolute-error columns are infinite; ULP metrics remain the primary finite comparison. * The searched-core fp32 row shows average-metric improvement: mean ULP and mean absolute error improve over $[0, 0.7]$, while worst-case ULP and worst-case absolute error worsen. It is not a global `tanhf` replacement claim. † The localized fp32 row is reported as a secondary local result: on $[0.3, 0.5]$, max absolute error drops by 23.2 percent while max ULP ties.

Table 7. Per-interval arithmetic size audit. Baseline counts are source-expanded estimates including the arithmetic inside `expf`/`expm1f`. Smaller is better.

Task	Ours: DAG ops by interval	Expanded baseline ops	Ours smaller by
sigmoid fp16	$[0, .5]: 5$, $[.5, 1]: 7$, $[1, 2]: 4$, $[2, 3]: 9$, $[3, 4]: 8$, $[4, 8]: 7$	$\approx$ 12–20	$\geq \mathbf{1.3}\times$
tanh fp16	$[0, .25]: 3$, $[.25, 1]: 6$, $[1, 5]: 9$	$\approx$ 19–33	$\geq \mathbf{2.1}\times$
sinh fp16, low/mid	$[0, 3]: 10$, $[3, 5]: 11$, $[5, 7]: 7$, $[7, 9]: 8$	$\approx$ 20–35	$\geq \mathbf{1.8}\times$
sinh fp16, high	$[9, 9.907]: 6$, $[9.907, 10.375]: 9$, $[10.375, 11]: 11$, $[11, 11.75]: 10$	$\approx$ 16–20	$\geq \mathbf{1.4}\times$
tanh fp32	$[0, .3]: 6$, $[.3, .5]: 7$, $[.5, .7]: 7$	$\approx$ 19–32	$\geq \mathbf{2.7}\times$

max ULP drops from 287 to 2. The max absolute error ties because fp16 output spacing near saturation dominates the worst absolute-error point; ULP exposes the quality difference more clearly. For sinh fp16, the eight-piece positive partition concentrates cuts in the upper range where fp16 intermediates lose dynamic range, reducing max ULP from 1021 to 7 and mean ULP from 2.798 to 0.208.

5. Conclusion

We have presented a precision-aware, agent-orchestrated pipeline for the automated design of elementary-function approximations. The pipeline composes three loops with cleanly separated responsibilities: an outer LLM agent that plans partitioning, routing, verification, and retry against a typed tool interface; a middle evolutionary search over DAGs adopted essentially unchanged from AutoNumerics-Zero; and an inner two-stage constant fitter whose Levenberg–Marquardt / Nelder–Mead pairing is justified by the empirical landscape of the constant-fitting objective. On 2^x DAGs from AutoNumerics-Zero our inner loop matches or beats the published fp64 optima while the original CMA-ES solver fails by orders of magnitude on the harder structures, isolating the contribution of the inner loop. A controller ablation similarly isolates the outer loop: under a fixed inner/middle loop and a common budget, the agent's partition attains lower fp16 worst-case ULP than fixed-global, fixed-uniform, and a non-LLM counterexample-driven controller, with fewer

and better-placed pieces. End-to-end, the pipeline produces fp16 sigmoid, tanh, and sinh kernels with substantially lower ULP error than pure-fp16 OpenLibm-style baselines.

Impact Statement

This paper presents work whose goal is to advance automated numerical software design. The methods target lower-error and more compact elementary-function kernels under explicit precision models. Potential societal consequences are therefore similar to those of improved scientific-computing and machine-learning infrastructure; we do not identify additional impacts that require specific highlighting.

References

Brisebarre, N. and Chevillard, S. Efficient polynomial l-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH'07)*, pp. 169–176. IEEE, 2007.

Chevillard, S., Joldeş, M., and Lauter, C. Sollya: An environment for the development of numerical codes. In *International congress on mathematical software*, pp. 28–31. Springer, 2010.

Daramy, C., Defour, D., De Dinechin, F., and Muller, J.-M. Cr-libm: A correctly rounded elementary function library. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, volume 5205, pp. 458–464. SPIE, 2003.

Hairer, E., Wanner, G., and Nørsett, S. P. *Solving ordinary differential equations I: Nonstiff problems*. Springer, 1993.

Hansen, N. and Ostermeier, A. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

Kupriianova, O. and Lauter, C. Metalibm: A mathematical functions code generator. In *International Congress on Mathematical Software*, pp. 713–717. Springer, 2014.

Lim, J. P., Aanjaneya, M., Gustafson, J., and Nagarakatte, S. An approach to generate correctly rounded math libraries for new floating point variants. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–30, 2021.

Marquardt, D. W. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

Muller, J.-M. *Elementary functions: algorithms and implementation*. Springer, 2006.

Nelder, J. A. and Mead, R. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

Oppenheim, A. V. *Discrete-time signal processing*. Pearson Education India, 1999.

Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. Automatically improving accuracy for floating point expressions. *Acm Sigplan Notices*, 50(6):1–11, 2015.

Real, E., Chen, Y., Rossini, M., de Souza, C., Garg, M., Verghese, A., Firsching, M., Le, Q. V., Cubuk, E. D., and Park, D. H. Autonumerics-zero: automated discovery of state-of-the-art mathematical functions. *arXiv preprint arXiv:2312.08472*, 2023.

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

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.