

Enhancing Mathematical Problem Solving in LLMs through Execution-Driven Reasoning Augmentation

Anonymous ACL submission

Abstract

Mathematical problem solving is a fundamental benchmark for assessing the reasoning capabilities of artificial intelligence and a gateway to applications in education, science, and engineering where reliable symbolic reasoning is essential. Although recent advances in multi-agent LLM-based systems have enhanced their mathematical reasoning capabilities, they still lack a reliably revisable representation of the reasoning process. Existing agents either operate in rigid sequential pipelines that cannot correct earlier steps or rely on heuristic self-evaluation that can fail to identify and fix errors. In addition, programmatic context can distract language models and degrade accuracy. To address these gaps, we introduce Iteratively Improved Program Construction (*IIPC*), a reasoning method that iteratively refines programmatic reasoning chains and combines execution feedback with the native Chain-of-thought abilities of the base LLM to maintain high-level contextual focus. *IIPC* surpasses competing approaches in the majority of reasoning benchmarks on multiple base LLMs. All code and implementations will be released as open source upon publication.

1 Introduction

Mathematical reasoning, the ability to reliably solve multi-step math problems through symbolic manipulation and language-based deduction, is a necessary skill for LLM-based systems to master in order to build generally intelligent systems that can automate intellectual tasks. It is especially crucial in applications such as scientific discovery (Cherian et al., 2024; Shojaee et al., 2025),

optimization (Huang et al., 2024; Forootani, 2025; Habeeb et al., 2023), financial modeling (Yao et al., 2025; Carvalho Santos et al., 2022), and education (Gupta et al., 2025; Liu et al., 2025), among many others. As research advances towards enabling language models to use tools and understand abstract concepts, mathematical reasoning emerges as a benchmark for assessing their ability to combine symbolic structure with logical deduction. Advancing this capability is necessary for building AI systems that can engage in effective, applicable, and scientifically grounded reasoning.

Despite substantial progress, many reasoning agents still struggle with two limitations. First, most systems lack a **persistent and reliably manipulable representation of their overall reasoning state** that allows them to make informed revisions of earlier steps. Once a model commits to a reasoning chain, whether expressed as text, a sequence of agentic steps, or a program, subsequent revisions are typically superficial, based on heuristic self-evaluation or nonexistent. Without an editable global reasoning state, the model cannot identify or act on contradictions, invalid computations, or revise previous steps, making it vulnerable to cascading errors (Cemri et al., 2025). Second, execution-guided agents lack **stabilizers against program bias, often over-prioritizing execution signals** that could be logically flawed over token-level reasoning, resulting in brittle reasoning trajectories. When program outputs are used to draw conclusions, irrelevant or incorrect information may cause biased reasoning in models, reducing accuracy (Shi et al., 2023; Yang et al., 2025).

Existing approaches address portions of these limitations, but leave important gaps. Multi-agent systems such as Cumulative Reasoning (CR) (Zhang et al., 2023) and Multi-agent Condition Mining (MACM) (Lei et al., 2024) have enabled stepwise deliberation. However, their sequential structure locks their reasoning trajectory into a fixed forward direction. This rigidity prevents agents from freely revising earlier steps, making them vulnerable to cascading errors, a phenomenon identified by (Cemri et al., 2025) as "inter-agent misalignment". Agents like Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), Tree-of-Thoughts (Yao et al., 2023), and Graph-of-Thoughts (Besta et al., 2024) offer self-evaluation, refinement, and backtracking capabilities, but their effectiveness is bounded by their capacity to produce accurate self-evaluations (Huang et al., 2023), often resulting in unreliable refinements and trajectory selections. Program and tool-based agents such as Program of Thoughts (PoT) (Chen et al., 2023), Tool Integrated Reasoning Agent (ToRA) (Gou et al., 2024), Code-based Self Verification (CSV) (Zhou et al., 2023), and Program-aided Language Model (PAL) (Gao et al., 2023) provide deterministic signals through code execution. These programs are usually one-off artifacts and are not subjected to targeted revisions, leading to rigidity similar to that of sequential reasoning agents. Furthermore, recent studies have demonstrated that such agents, particularly those relying on program-based reasoning, are susceptible to irrelevant or misleading context, often over-incorporating flawed information into their reasoning traces and causing performance degradation (Shi et al., 2023; Yang et al., 2025).

To overcome these limitations, we propose Iteratively Improved Program Construction (IIPC), an agent that treats programs as explicit representations of the model’s reasoning chain. While most agents generate one-off code blocks, IIPC generates reasoning chains as programs designed for transparent inspection and evolution through informed revision. Each revision is grounded in deterministic feedback that includes information about intermediate steps, allowing the model to correct errors or inconsistencies in its reasoning and make causally informed changes. A memory of past mistakes is maintained to ensure that refinements minimize "revisit regret" and steer improvements away from failure modes rather than resampling failed programs. To preserve a high-

level contextual focus, IIPC adopts a dual-branch architecture. The token-level reasoning branch produces a CoT reasoning trace free from dependence on program output, while the program-refinement branch iteratively refines an executable representation of the model’s reasoning. The outputs of the two branches are combined only at the final stage to provide context for the final reasoning chain. Through this mechanism, IIPC avoids over-reliance on potentially incorrect or irrelevant program results while leveraging relevant information. As a result, IIPC provides a unified reasoning mechanism that combines manipulable representations of reasoning traces with context-stable reasoning, overcoming the limitations of existing multi-agent, self-evaluating, and execution-guided systems.

The paper’s primary contributions are as follows:

1. We introduce IIPC, a new reasoning method that is designed to refine programs through execution-guided feedback, integrate execution outputs into its own reasoning abilities, and surpass other code-based, state-of-the-art, non-ensemble reasoning agents in difficult math problem-solving benchmarks.
2. We provide the reasoning-trace corpus generated from testing IIPC, including problem statements, initial propositions, generated code, execution outputs, integrated deliberation, and final answers, to facilitate reproducible evaluation, detailed error analysis, and future work on program-centric reasoning.
3. We conduct a comprehensive evaluation of IIPC against various multi-agent reasoning methods across large language models and mathematical reasoning benchmarks. IIPC surpasses PoT, MACM, and CR on the majority of the mathematical problem-solving benchmarks on multiple base LLMs.

2 Related Work

Recently developed multi-agent systems vary in their approach to structuring mathematical reasoning. CR (Zhang et al., 2023) uses three components: a proposer, verifier, and reporter. These components generate, evaluate, and collect reasoning steps. CR achieves 72.2% accuracy on 500 MATH problems (Hendrycks et al., 2021) using GPT-4 and early LLaMA models. MACM (Lei et al., 2024) accumulates and verifies conditions needed to solve the given problem rather than processing

172	the reasoning steps. MACM also uses voting mechanisms to leverage uncertainty and explore multiple reasoning trajectories that are higher in quality than CoT (Wei et al., 2022), Tree-of-Thoughts (ToT) (Yao et al., 2023), or Graph-of-Thoughts (GoT) (Besta et al., 2024) prompting. Despite their improvements on popular benchmarks, sequential processing introduces rigid reasoning chains that cause errors to propagate over subsequent steps.	224
173		225
174		226
175		227
176		228
177		229
178		230
179		231
180		232
181	While sequential multi-agent systems like MACM and CR focus on reasoning through iterative step accumulation, other research directions enhance problem-solving by integrating external tools. Tool-Integrated Reasoning Agent (ToRA) (Gou et al., 2024) fine-tunes LLaMA models to interleave natural language reasoning with tool calls. PoT (Chen et al., 2023) generates and executes Python programs, using the outputs to arrive at the final answer. Code-Based Self-Verification (CSV) (Zhou et al., 2023) includes programmatic checks to verify and refine solutions, improving reliability through iterative code-driven validation.	233
182		234
183		235
184		236
185		237
186		238
187		239
188		240
189		241
190		242
191		243
192		244
193		245
194	Other approaches aim to improve reasoning internally through self-correction. Agents like self-refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) adopt an iterative self-improvement approach, generating feedback on their own output and refining it until a predefined stopping condition is met. These methods provide foundational insights for enhancing program-based reasoning, which our approach leverages and advances.	246
195		247
196		248
197		249
198		250
199		251
200		252
201		253
202		254
203	IIPC’s program refinement mechanism is similar to CodePRM (Li et al., 2025) in its usage of execution feedback to refine reasoning. However, IIPC treats programs as representations of the reasoning chain, whereas CodePRM edits natural-language thoughts based on their quality scores for better downstream execution. The dual-branch architecture used by IIPC conceptually resembles the Talker-Reasoner architecture (Christakopoulou et al., 2024), where a talker agent performs fast language generation, the reasoner performs multi-step reasoning, and they interact through a shared memory. However, unlike the interleaved process of the Talker-Reasoner, IIPC maintains a clean CoT trace and only merges at the end to remain robust to noisy programmatic outputs.	255
204		256
205		257
206		258
207		259
208		260
209		261
210		262
211		263
212		264
213		265
214		266
215		267
216		268
217		269
218		270
219	Execution-Guided Classifier-Free Guidance (EG-CFG) (Lavon et al., 2025) is a decoding strategy that guides token sampling based on line-level execution. OpenCodeInterpreter (OCI) (Zheng et al., 2024) uses an execution guided refinement loop to iteratively refine generated programs. Outcome-Refining Process Supervision (ORPS) (Yu et al., 2025) performs a beam search over reasoning-code trajectories, selecting candidates based on execution-based outcome signals. In contrast, IIPC treats programs as representations for mathematical problem solving, refining them using both process- and outcome-level signals to identify and correct reasoning errors.	271
220		
221		
222		
223		

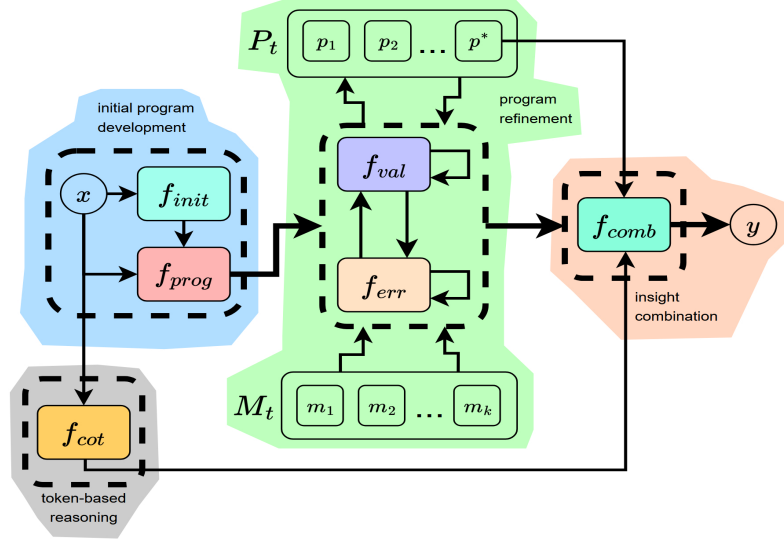


Figure 1: Overview of IIPC. f_{init} derives key propositions from the problem statement; f_{prog} generates an initial candidate program; f_{val} evaluates program correctness and logical consistency; if errors are detected, the error correction component f_{err} revises the program accordingly; f_{cot} produces a textual chain of thought; f_{comb} combines program and token reasoning context for final output; M_t denotes the error descriptor memory at refinement step t ; P_t represents the program store at step t .

to evaluate its correctness.

Once the program is constructed, it is executed using an interpreter E to yield an output ($o_1 = E(p_1)$), which can either be the program’s output, an error, or both. This serves as the basis for subsequent reflection and refinement.

IIPC reviews the generated program’s reasoning and consults a persistent reflection memory containing past mistakes (M_t , where t is the time point of the current refinement step) to avoid repeating unsuccessful strategies. It then either (1) issues a revised program accompanied by a concise reflection on the identified flaw, which is logged in the memory store before the revised program is re-executed, or (2) outputs a stopping signal indicating that the current program and its outputs are satisfactory for solving the problem without further refinement. If the program produces an error, IIPC amends only the offending code segment without modifying the reasoning chain itself. It also produces a concise reflection on the mistake, which is appended to its memory store for guiding future iterations.

IIPC iteratively refines the design by executing a series of process validations and error corrections to produce the optimal program for solving the given problem. If a program does not result in an error, the *process validation* component will assess the program’s validity and its output. However, if the program produces an error, the *error correction*

component will revise it while preserving the reasoning chain. The resulting programs from either process are rerun and cycled (along with their output and error) back into one of the two components for further correction and/or enhancement. The current implementation for refinement, as tested in the main experiments, allows a maximum of two process validations and two error corrections after each process validation. All working programs and their output are stored in an array P_t , with the program with the highest index being the most recently developed working program. This back-and-forth loop between the process validation and error-correction components continues until a stop condition is met. The *stop condition* could either be a satisfactory program, a limit on the allowed number of refinements or error corrections, or a limit on the token usage of the agent. Formally, this refinement process can be summarized in the following formulas:

$$(p_{t+1}, m_{t+1}) = \begin{cases} f_{err}(x, s, p_t, o_t, M_t), & \text{if } o_t \in e, \\ f_{val}(x, s, p_t, o_t, M_t), & \text{if } o_t \notin e \end{cases} \quad (1)$$

$$M_{t+1} = M_t \cup m_{t+1} \quad (2)$$

$$P_{t+1} = P_t \cup p_t \text{ if } o_t \notin e \quad (3)$$

where m_{t+1} is an error or mistake descriptor, f_{err} is the function that corrects errors, and f_{ref}

is the refinement function that can result in a \emptyset for both p_{t+1} and m_{t+1} if the refinement process reaches a stopping point. In addition to program construction and refinement, IIPC generates a purely text-based CoT c in a separate branch and arrives at a temporary answer ($c = f_{cot}(x, s)$). The most recent working program ($p^* \in P_t$) and its output (o^*) with the generated CoT (c) text is concatenated through a structured integration prompt, which is used as input to the underlying LLM to arrive at the final answer for the problem ($y = f_{comb}(x, s, p^*, o^*, c)$). This step allows IIPC to reevaluate the solution using symbolic evidence from both refined program execution and native linguistic reasoning, thereby reducing over-dependence on a single kind of context.

4 Experiments

Experimental Setup. Our evaluation benchmarks the IIPC framework on two mathematical reasoning datasets: (1) MATH (Hendrycks et al., 2021), comprising complex problems in multiple subjects with five difficulty levels; (2) and AIME (American Invitational Mathematics Examination) (Veeraboina, 2024), consisting of difficult competition math problems from 1983 to 2024 that emphasize creative problem solving and is more challenging than the MATH dataset. For the MATH dataset, we evaluate on a balanced subset of the official test split from the original dataset, targeting uniform coverage across all topic-difficulty combinations (35 bins total). Due to limited availability in certain bins, the final subset contained 1483 problems total. For the AIME dataset (Veeraboina, 2024), we use the complete set of 933 problems spanning the years 1983-2024. We compare IIPC’s performance against other reasoning paradigms, including CR (Zhang et al., 2023), MACM (Lei et al., 2024), and PoT prompting (Chen et al., 2023). We evaluate across multiple state-of-the-art LLMs, including GPT-4o mini, Gemini 2.0 Flash, Mistral Small 3.2 24B, Gemma 3 27B, and Llama 4 Maverick, to characterize how model architecture influences accuracy. Finally, our ablations isolate the contributions of specific architectural components, examine the effect of varying decoding temperatures, evaluate all methods under voting-based aggregation, and use GSM8K (Cobbe et al., 2021), a data set consisting of grade school math problems, to assess performance degradation due to agentic overhead on simpler math problems.

Agent Implementations. For the purposes of this study, the main evaluation tests all prompting-based reasoning agents with access to code interpreters, excluding multi-trajectory aggregation (voting) mechanisms due to their high token usage and costs. To comply with this constraint, reasoning agents like MACM (Lei et al., 2024), which utilize and aggregate multiple answers, are adapted to operate within a single run and evaluated accordingly. While this may handicap the MACM agent, which is designed to operate in an ensemble environment, testing its performance in these conditions allows for a fair comparison with other such agents and provides insights into the weaknesses of processing steps sequentially rather than in parallel. As mentioned before, a separate ablation is provided to assess the effects of voting environments on the performance of all agents. Our implementation of PoT includes an error-correction loop that retries program generation when execution fails, until it reaches a fixed retry limit and has to fall back to standard Chain-of-Thought reasoning. This design allows us to better compare the mathematical reasoning capabilities of agents against PoT while controlling for coding ability. Our custom implementations of MACM and CR attempt to incorporate as much of the original implementations as possible and follow the proposed methods closely (with the exception of the voting mechanism for MACM). All agents are tested with an LLM decoding temperature of 0.1 and measure performance as accuracy (the number of correctly solved questions divided by the total number of questions evaluated).

Assessing Correctness. To assess answer correctness, we first use deterministic equivalence and fall back to LLM-based judging using the LLaMA 4 Maverick model if necessary. Although deterministic evaluation is ideal, code-centric agents may produce equivalent answers in differing formats (e.g., 1.4142 versus $\sqrt{2}$). To ensure transparency and reproducibility, we fix the decoding temperature at 0 for the LLM judge and release the full judging code. On the AIME dataset, 97.78% of answers were verified via exact integer matching. On the MATH dataset, 72.85% of the generated answers were deterministically verified; LLM judging was only used to confirm whether the answers were incorrect after deterministic judging failed. On GSM8K, 89.35% of the answers were verified using exact answer matching.

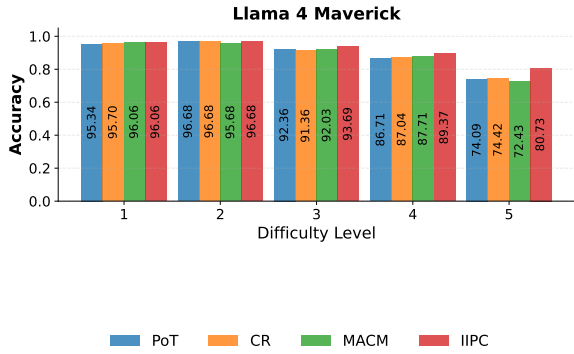


Figure 2: Accuracy of PoT, IIPC, CR, and MACM on the MATH dataset using Llama 4 Maverick. This bargraph is stratified by difficulty level

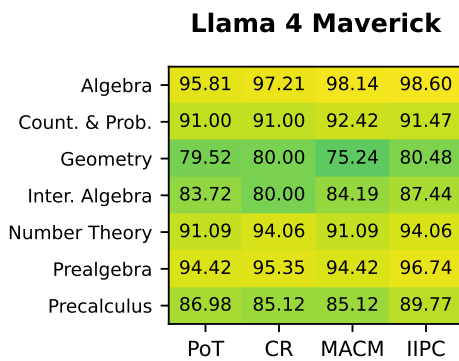


Figure 3: Heatmap of accuracy (%) by subject area on the MATH benchmark for Llama-4-Maverick. Columns correspond to reasoning agents (PoT, IIPC, CR, and MACM) and rows correspond to mathematical domains.

Other Considerations. While our evaluations focus on accuracy and tend to favor reasoning stability over token efficiency, we do recognize the fundamental limitation that IIPC incurs significant token costs by regenerating refined or corrected programs with every iteration. Additionally, our evaluation reports single-run accuracy on fixed benchmarks. However, while gains may be small for some evaluations, IIPC consistently outperforms other agents across all LLMs and datasets, reducing the likelihood that performance gains are due to noise.

4.1 MATH

The MATH benchmark includes problems across a broad spectrum of advanced topics, including number theory, geometry, algebra (from prealgebra through intermediate algebra), precalculus, and counting & probability, each presented at five increasing difficulty tiers. On this benchmark, IIPC outperformed the other reasoning methods across most LLM backbones, achieving the high-

est scores on Gemini 2.0 Flash (94.13%), Mistral Small 3.2 24B (90.83%), Gemma 3 27B (90.56%), and Llama 4 Maverick (91.23%). Conversely, PoT surpassed IIPC and all other methods on GPT-4o-mini (81.19%). This suggests that, in models with lower reasoning capacity, the additional complexity introduced by IIPC can become counterproductive, whereas PoT reasoning traces appear better aligned with these models' capabilities, providing just enough scaffolding without overwhelming their reasoning bandwidth.

To further analyze agent behavior on the MATH benchmark, we break down performance by difficulty level and topic. Although the main evaluation presents this analysis only for Llama 4 Maverick, analogous results for all other language models are provided in the appendix. As evident in figure 2, IIPC consistently matches or outperforms other agents as difficulty increases. On the most challenging level-5 problems, IIPC achieves an accuracy of 80.73, outperforming the next best method by 6.31 points. As shown in figure 3, we find that with the exception of Counting and Probability, IIPC achieves the highest accuracy across all domains. Most notably, we record the highest gains in Pre-calculus (+2.79%, 89.77%) and Intermediate Algebra (+3.25%, 87.44%). This provides further evidence for IIPC's ability to solve difficult problems and its effectiveness over existing methods.

4.2 AIME

The AIME dataset contains problems from the American Invitational Mathematics Examination from 1983 to 2024. It emphasizes creative problem-solving and is the most difficult of the 3 datasets tested. Consistent with its performance on the MATH benchmark, IIPC achieved the highest accuracy on most models, specifically Gemini 2.0 Flash (64.20%), Mistral 3.2 24B (52.52%), Gemma 3 27B (50.48%), and Llama 4 Maverick (69.77%). In contrast, PoT surpassed all other methods on GPT-4o-mini (31.40%). Similar to the results on the MATH benchmark, GPT-4o-mini may lack the reasoning capacity required to fully benefit from IIPC's more refined loop. IIPC's design scales well to highly challenging mathematical reasoning tasks when supported by sufficiently capable models, while PoT seems to remain more robust on models with more constrained reasoning capacity.

Table 1: Accuracy (%) of reasoning methods across five LLMs. IIPC row is shaded; bold teal = per-column SOTA. Numbers in parentheses show Δ vs PoT baseline.

MATH					
Reasoning System	GPT-4o-mini	Gemini 2.0 Flash	Mistral 3.2 24B	Gemma 3 27B	Llama 4 Maverick
PoT	81.19	92.58	89.62	89.01	88.94
CR	76.53 (-4.66)	90.09 (-2.49)	83.61 (-6.01)	87.05 (-1.96)	89.94 (+1.00)
MACM	72.62 (-8.57)	90.09 (-2.49)	82.13 (-7.49)	86.72 (-2.29)	88.67 (-0.27)
IIPC	80.98 (-0.21)	94.13 (+1.55)	90.83 (+1.21)	90.56 (+1.55)	91.23 (+2.29)

AIME					
Reasoning System	GPT-4o-mini	Gemini 2.0 Flash	Mistral 3.2 24B	Gemma 3 27B	Llama 4 Maverick
PoT	31.40	59.16	48.12	46.20	62.49
CR	23.90 (-7.50)	53.48 (-5.68)	40.09 (-8.03)	41.69 (-4.51)	62.17 (-0.32)
MACM	17.79 (-13.61)	51.98 (-7.18)	37.62 (-10.50)	41.69 (-4.51)	62.17 (-0.32)
IIPC	29.05 (-2.35)	64.20 (+5.04)	52.52 (+4.40)	50.48 (+4.28)	69.77 (+7.28)

Table 2: Accuracy (%) of reasoning methods on GSM8K. Bold teal = per-row SOTA. Numbers in parentheses show Δ vs CoT baseline.

LLM	CoT	IIPC	PoT	MACM	CR
GPT-4o-mini	94.54	94.24	93.56	91.05	92.34
Gemini 2.0 Flash	96.06	95.60	95.75	92.57	91.89
Mistral 3.2 24B	95.22	95.00	94.84	93.86	94.47
Gemma 3 27B	95.45	94.84	94.47	91.36	93.18
Llama 4 Maverick	96.13	95.53	95.53	94.24	95.91

Table 3: Ablation results on MATH (Gemini 2.0 Flash) and AIME (Gemini 2.0 Flash, GPT-4o-mini). Best per column is **bold teal**.

Methods	MATH (Gemini)		AIME	
	Gemini 2.0 Flash	Gemini 2.0 Flash	Gemini 2.0 Flash	GPT-4o-mini
CoT	92.92	56.70	21.11	
PoT-NC	92.52	59.12	31.19	
PoT	92.58	59.16	31.40	
IIPC-NS-NMS	93.19 (+0.61)	60.77 (+1.61)	26.90 (-4.50)	
IIPC-NS	93.59 (+1.01)	61.52 (+2.36)	27.12 (-4.28)	
IIPC	94.13 (+1.55)	64.20 (+5.04)	29.05 (-2.35)	

4.3 Ablation Tests

To better understand which components of IIPC contribute most to its performance gains, we conducted a series of ablation studies on both the MATH and AIME benchmarks. The first set of ablations incrementally adds architectural features across five methods:

- CoT: A baseline Chain-of-Thought method with purely token-level reasoning.
- PoT-NC: A Program-of-Thoughts method that generates and executes programs but omits the correction loop.

Table 4: IIPC temperature ablation on AIME (Gemini 2.0 Flash). Best is **bold teal**.

Temperature	Accuracy (%)
0.1	64.20
0.3	64.52
0.5	63.56
0.7	63.13
0.9	63.24

Table 5: Effect of voting on MATH Level-5 (245 problems) using Llama 4 Maverick. Best in each column is **bold teal**.

Agent	No Voting	Voting
CoT	70.61	74.29
PoT	73.47	80.30
CR	68.57	78.37
MACM	69.80	76.33
IIPC	78.78	80.30

- PoT: A PoT method with both program generation and execution-guided error correction.
- IIPC-NS-NMS: An IIPC variant with the full correction and iterative refinement loop but without the token-reasoning branch or persistent reflection memory. Conclusions are made directly from the programmatic context.
- IIPC-NS: An IIPC variant with the full correction and iterative refinement loop without the token-reasoning branch, but with the persistent reflection store. Conclusions are made directly from the programmatic context.
- IIPC: The full IIPC reasoning agent.

Architectural Ablation Analysis. Table 3 summarizes the impact of key IIPC mechanisms across datasets and model backbones. On Gemini 2.0 Flash, the iterative refinement, reflection memory, and dual-branch separation provide substantial gains. Iterative refinement improves accuracy by +0.61% on the MATH dataset (92.58 \rightarrow 93.19) and +1.61% on the AIME dataset (59.16 \rightarrow 60.77). The reflection memory improves accuracy by an additional 0.4% on the MATH dataset (93.19 \rightarrow 93.59) and 0.75% on the AIME dataset (60.77 \rightarrow 61.52). Dual-branch separation yields an additional +0.54% on the MATH dataset (93.59 \rightarrow 94.13) and +2.68% on the AIME dataset (61.52 \rightarrow 64.20). Together, these mechanisms increase performance by 1.55% over PoT on the MATH dataset (92.58 \rightarrow 94.13), and by 5.04% over PoT on the AIME dataset (59.16 \rightarrow 64.20), highlighting their effectiveness for tasks requiring structured reasoning. On GPT-4o mini, while PoT achieves the highest accuracy, ablations show that individual IIPC components partially mitigate the performance gap. The reflection store and dual-branch separation reduce the accuracy drop of IIPC relative to PoT, even though the accuracy of the full IIPC agent remains below the PoT baseline.

Complexity Ablation Analysis. We also compared IIPC, MACM, CR, and PoT against CoT on GSM8K, as shown in Table 2, to determine whether agent complexity hinders accuracy compared to chain of thought reasoning on easier math problems. CoT performs near the upper limit across most models, achieving 95–97% accuracy on the strongest backbones. On average, IIPC and PoT closely trail CoT’s performance, while MACM and CR have slightly lower accuracy. These results suggest that GSM8K’s problem structure saturates the benefits of iterative refinement, leaving little room for improvement beyond direct token-level reasoning on stronger models. Nonetheless, IIPC’s close performance to CoT demonstrates that its structure does not significantly degrade accuracy, even in low-complexity reasoning environments.

Temperature Ablation Analysis. We further examined the effect of varying the decoding temperature on IIPC’s performance with Gemini 2.0 Flash on the AIME benchmark (Table 4). Across the tested range (0.1–0.9), accuracy remained relatively stable, fluctuating within a narrow band of 1.39 percentage points. The best performance was achieved at 0.3 (64.52%), while higher tem-

peratures (0.5–0.9) yielded slightly lower accuracies (63.13%–63.56%). These results suggest that IIPC is robust to temperature changes, with only marginal sensitivity, although higher temperatures appear to slightly diminish performance.

Voting Ablation Analysis. Beyond architectural and temperature-based variations, we also evaluate the effect of voting, a method used in the original MACM implementation, on IIPC’s performance. As detailed in the appendix, our evaluation excludes stochastic multi-trajectory aggregation due to high costs, but we isolate its effect in a smaller ablation study using Llama-4-Maverick on 245 Level-5 MATH problems (35 from each topic). To align with MACM’s original setting, we use a decoding temperature of 0.1 for the non-voting environment, 0.7 for the voting environment, a minimum of 5 voters, and a maximum of 7 voters for tie-breaking. The results in Table 5 show that without voting, IIPC achieves the highest accuracy at 78.78%, outperforming even voting-based variants of many baselines. When voting is enabled, IIPC achieves the same accuracy as PoT at 80.30%. Voting yields larger gains for PoT (+6.83%), CoT (+3.68%), CR (+9.8%), and MACM (+6.53%), while the relative improvement for IIPC (+1.52%) is smaller. IIPC exhibits smaller gains from voting, suggesting reduced single-trajectory variance compared to baselines that rely more on sampling diversity.

5 Conclusion

In this work, we introduced Iteratively Improved Program Construction (IIPC) for mathematical reasoning with LLMs through execution-guided refinement while mitigating execution bias by reducing over-dependence on program-based context. Our experiments show that IIPC consistently delivers competitive or superior performance. On GSM8K, where simpler reasoning tasks leave limited room for improvement, IIPC demonstrated that its additional structure does not significantly hinder its reasoning capabilities compared to simpler baselines. On both the MATH and AIME datasets, IIPC achieved the highest accuracy across 4 of the 5 models tested. These results demonstrate that integrating token-based reasoning with iterative program refinement provides LLMs with valuable capacity for trajectory correction and reasoning-level regularization, reducing over-commitment to execution-conditioned context.

6 Limitations

Despite the advantages of IIPC, our results also revealed some trade-offs. On GPT-4o mini, IIPC lagged behind PoT on both MATH and AIME. These findings suggest that while IIPC scales effectively with models capable of sustaining its iterative refinement loop, models with limited reasoning or coding abilities may instead benefit from PoT’s simpler structure. In addition, although IIPC’s design enables advanced refinement for solving difficult problems, it is a token-intensive approach to reasoning. While IIPC proves robust on GSM8K and state-of-the-art on MATH and AIME across several high-capacity models, there is room for improvement in subsequent research. Future directions include addressing token efficiency in IIPC’s design, adapting to models’ reasoning capacity, and extending IIPC to other domains requiring verifiable, structured reasoning beyond mathematics.

7 Ethical Considerations

Our work introduces an execution-guided agent for enhanced mathematical reasoning in LLMs. The proposed method is evaluated on publicly available benchmarks (AIME, MATH, GSM8K) and does not involve human subjects, personal data, or proprietary datasets. We have ensured that none of the data used or generated by this research contains information that uniquely identifies people. While IIPC is designed to improve reasoning capabilities, it doesn’t guarantee correctness. Any outputs generated by IIPC should not be relied on without independent verification, especially in high-stakes or real-world decision making contexts.

ChatGPT was used to provide feedback on quality, assist with some grammatical refinement and articulation, and support understanding and exploration of existing mathematical concepts for framing the proposed method. However, all revisions were carefully considered and manually done. All ideas, experimental designs, analyses, and conclusions were developed by the authors, who take full responsibility for the contents of the paper. The judging agent codebase, originally written by the authors, was later enhanced with ChatGPT to improve robustness and evaluation reliability, and was reviewed and verified by the authors.

The proposed method is intended for research, benchmarking, and assistive applications. It is not meant to replace human judgement or facilitate academic dishonesty. To promote transparency

and reproducibility, all code, evaluations, sampled datasets, and reasoning traces generated by the proposed method are released as open source. We hope that our work will contribute positively to ongoing research in LLM reasoning.

References

- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and 1 others. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17682–17690.
- Gustavo Carvalho Santos, Flavio Barboza, Antônio Veiga, and Kamyrr Gomes. 2022. *Portfolio optimization using artificial intelligence: A systematic literature review*. *Exacta*, 0.
- Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, and 1 others. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. *Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks*. *Preprint*, arXiv:2211.12588.
- Anoop Cherian, Radu Corcodel, Siddarth Jain, and Diego Romeres. 2024. *Llmphy: Complex physical reasoning using large language models and world models*. *Preprint*, arXiv:2411.08027.
- Konstantina Christakopoulou, Shibl Mourad, and Maja Matarić. 2024. *Agents thinking fast and slow: A talker-reasoner architecture*. *Preprint*, arXiv:2410.08328.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.
- Ali Forootani. 2025. *A survey on mathematical reasoning and optimization with large language models*. *Preprint*, arXiv:2503.17726.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. *Pal: Program-aided language models*. *Preprint*, arXiv:2211.10435.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujia Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2024. *ToRA: A tool-integrated reasoning agent for mathematical problem solving*. In *The*

724		Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning . <i>Preprint</i> , arXiv:2303.11366.	779
725			780
726	Adit Gupta, Jennifer Reddig, Tommaso Calo, Daniel Weitekamp, and Christopher J. MacLellan. 2025. Beyond final answers: Evaluating large language models for math tutoring . <i>Preprint</i> , arXiv:2503.16460.		781
727			782
728		Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. 2025. Llm-sr: Scientific equation discovery via programming with large language models . <i>Preprint</i> , arXiv:2404.18400.	783
729			784
730	Hiyam Adil Habeeb, Dzuraidah Abd Wahab, Abdul Hadi Azman, and Mohd Rizal Alkahari. 2023. Design optimization method based on artificial intelligence (hybrid method) for repair and restoration using additive manufacturing technology . <i>Metals</i> , 13(3).		785
731			786
732		Hemish Veeraboina. 2024. Aime problem set (1983–2024). https://www.kaggle.com/datasets/hemishveeraboina/aime-problem-set-1983-2024/data . Accessed October 2025.	788
733			789
734			790
735			791
736	Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. <i>NeurIPS</i> .		792
737			793
738		Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. <i>Advances in neural information processing systems</i> , 35:24824–24837.	794
739			795
740	Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. <i>arXiv preprint arXiv:2310.01798</i> .		796
741			797
742			798
743		Minglai Yang, Ethan Huang, Liang Zhang, Mihai Surdeanu, William Wang, and Liangming Pan. 2025. How is llm reasoning distracted by irrelevant context? an analysis using a controlled benchmark . <i>Preprint</i> , arXiv:2505.18761.	799
744			800
745	Sen Huang, Kaixiang Yang, Sheng Qi, and Rui Wang. 2024. When large language model meets optimization . <i>Preprint</i> , arXiv:2405.10098.		801
746			802
747			803
748	Boaz Lavon, Shahar Katz, and Lior Wolf. 2025. Execution guided line-by-line code generation . <i>Preprint</i> , arXiv:2506.10948.		804
749			805
750			806
751	Bin Lei, Yi Zhang, Shan Zuo, Ali Payani, and Caiwen Ding. 2024. Macm: Utilizing a multi-agent system for condition mining in solving complex mathematical problems . <i>Preprint</i> , arXiv:2404.04735.		807
752			808
753			809
754			810
755	Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025. CodePRM: Execution feedback-enhanced process reward model for code generation . In <i>Findings of the Association for Computational Linguistics: ACL 2025</i> , pages 8169–8182, Vienna, Austria. Association for Computational Linguistics.		811
756			812
757			813
758			814
759			815
760			816
761			817
762	Ben Liu, Jihan Zhang, Fangquan Lin, Xu Jia, and Min Peng. 2025. One size doesn't fit all: A personalized conversational tutoring agent for mathematics instruction . <i>Preprint</i> , arXiv:2502.12633.		818
763			819
764			820
765			821
766	Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhunoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback . <i>Preprint</i> , arXiv:2303.17651.		822
767			823
768			824
769			825
770			826
771			827
772			828
773			829
774	Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context . <i>Preprint</i> , arXiv:2302.00093.		830
775			831
776			832
777			833
778			834
		Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement . https://arxiv.org/abs/2402.14658 .	822
			823
			824
			825
			826
		Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification . <i>Preprint</i> , arXiv:2308.07921.	827
			828
			829
			830
			831
			832

A Theoretical Overview

A.1 Error Propagation in Sequential Agents

Sequential reasoning agents generate solution steps in an ordered manner, where each step s_t depends on prior steps $(s_1, s_2, \dots, s_{t-1})$. We can write this as:

$$s_t = f(s_1, s_2, \dots, s_{t-1}),$$

where f is the language model. Once a step s_n , with $1 \leq n \leq t-1$, has been completed, it remains permanent and cannot be revised. As a result, the conditional probability that the next step is correct, $\mathbb{P}(s_t \mid s_{1:t-1})$, reduces as errors accumulate in earlier steps. This causes initial inaccuracies to propagate through the rest of the reasoning trajectory without any opportunity for correction.

A.2 Iterative Program Refinement and Reflection Memory

Let \mathcal{P} denote the set of all candidate programs and $I : \mathcal{P} \rightarrow \mathcal{O}$ be the interpreter that produces an output from a program (including errors) $o = I(p)$. At refinement step t , the agent has a memory M_t that summarizes past mistakes or errors.

The IPC refinement operator is then

$$p_{t+1} = f_{\theta}(x, p_t, o_t, M_t) \quad (4)$$

$$= \arg \min_{p \in \mathcal{N}(p_t)} \left(\underbrace{\mathcal{L}(I(p))}_{\text{execution loss}} + \lambda \underbrace{\Psi(p; M_t)}_{\text{memory penalty}} \right). \quad (5)$$

where $\mathcal{N}(p_t)$ is the complete set of edits around p_t , o_t is the execution output at step t (programmatically output and errors), $\Psi(\cdot; M_t) \geq 0$ penalizes parts of the program that M_t should prevent, $\mathcal{L}(I(\cdot))$ penalizes the output of the program for undesirable outputs, and $\lambda > 0$ is a trade-off factor between fixing the current error and not repeating past mistakes. Intuitively, the language model attempts to find the next best program so that it produces desirable output and doesn't repeat past mistakes. We do emphasize that this objective is not explicitly optimized, but rather serves as a conceptual abstraction of how execution feedback and reflection memory bias the LLM's refinement behavior.

After developing p_{t+1} , the memory updates with the errors of the previous iteration as

$$M_{t+1} = M_t \cup \text{Desc}(p_t, I(p_t))$$

where Desc extracts failure descriptions (e.g. banned mathematical methods, reflections on mi-

nor errors, API misuse, etc) that are then appended to M_t .

A.3 Search and Regret in Refinement

Let $\mathcal{U}(M_t) \subset \mathcal{P}$ be the union of possible program regions avoided by the memory store of past mistakes. According to the results in table 3 where we isolate the effects of individual IPC components, the reflection memory does improve the accuracy of the IPC agent. Motivated by these results, we can model the LLM's refinement behavior under the penalty function $\Psi(\cdot; M_t)$ as follows: (1) it lowers the probability of generating programs from $\mathcal{U}(M_t)$, (2) it does so by at least some multiplicative factor whenever a program is a flawed, and (3) the LLM maintains diversity in its refinements to explore candidates outside $\mathcal{U}(M_t)$. Under these assumptions, we can qualitatively model the LLM's refinement behavior such that if $\Psi(\cdot; M_t)$ assigns a strictly positive penalty on $\mathcal{U}(M_t)$ and zero otherwise, then the expected probability of revisiting blacklisted regions decreases with t . As a result, the cumulative revisit regret

$$R_T = \sum_{t=1}^T \mathbf{1}\{p_t \in \mathcal{U}(M_{t-1})\}$$

grows sub-proportionally compared to naive refinement. Therefore, the expected per-step revisit rate decreases as memory accumulates.

Intuitively, this means that each time the LLM encounters an error, the memory store shifts the probability mass away from that type of failure, making it less likely to be chosen again. Over time, the agent spends a smaller fraction of steps repeating past mistakes, while still preserving the ability to explore new refinements within the edit space $\mathcal{N}(p_t)$. In other words, reflection memory turns trial-and-error into trial-and-improvement, where the system learns from its entire history of failures for a given problem.

A.4 Dual-Branch Separation

The objective of the dual-branch design is to find an answer y^* that is consistent with the information from both branches.

Let x denote the given problem, p the refined program to solve the problem, and o the output of p .

We can first model the reasoning behavior of the two branches in terms of the implicit distributions over candidate answers y as shown:

- $d_P(y|x, p, o)$: The distribution over candidate answer y conditioned on x , p , and o in the execution-guided reasoning branch.
- $d_T(y|x)$: The distribution over candidate answer y conditioned on just x in the token-based reasoning branch.

In order to reconcile these two potentially conflicting beliefs over the answer space, we introduce an idealized distribution d^* that takes into account information from both branches. We also introduce a weighting factor α that implicitly reflects how much trust should ideally be placed on the execution-guided branch relative to the token-based reasoning branch. The distribution d^* should remain as consistent as possible with both d_T and d_P , while also accounting for α . We can formalize this notion through a conceptual disagreement-penalized objective:

$$d^* = \arg \min_d \left[(1 - \alpha) D_{\text{KL}}(d \| d_T) \right. \quad (6)$$

$$\left. + \alpha D_{\text{KL}}(d \| d_P) \right]. \quad (7)$$

where $\alpha \in [0, 1]$ and D_{KL} is the Kullback-Leibler divergence. This objective penalizes solutions that are strongly supported by only one source of reasoning, unless that source is explicitly prioritized by α . Intuitively, this means that under d^* , answers that contradict token-level logical constraints or execution-based evidence receive a low probability, while answers reinforced by both branches receive a higher probability. It is important to note that while IIPC does not explicitly compute or optimize this objective, the dual-branch design induces behavior that resembles this form of regularization.

Finally, while d^* serves as a conceptual abstraction for the ideal distribution, we ultimately want an answer y^* that is viewed as the most consistent candidate under the reconciled distribution. This can be formalized through the following objective:

$$y^* = \arg \max_y d^*(y).$$

This abstraction shows how dual-branch separation stabilizes reasoning by producing answers that achieve consistency across both token-level and execution-based reasoning.

A.5 Effects of Model Capacity on Performance

Our results show IIPC is strongest on higher-capacity models (Gemini 2.0 Flash, Mistral 24B). This can be theoretically understood as an alignment between model capacity and refinement overhead. If the base LLM’s reasoning capacity is insufficient, the iterative correction signal increases cognitive load beyond what the model can process or was trained for, leading to lower accuracy.

B Additional Experimentation Details

Reasons for Dataset Selection. To capture a range of mathematical complexity, the main experiments evaluate IIPC on the MATH and AIME datasets, while the GSM8K dataset is used for ablation experiments to study effects of agentic overhead on simple problems. GSM8K (Cobbe et al., 2021) contains grade school arithmetic problems and serves as a low-difficulty benchmark for an ablation experiment to understand how additional complexity, such as IIPC’s refinement loop and dual-branch design, affects performance on simple tasks. The MATH dataset (Hendrycks et al., 2021) provides tests for algebraic manipulation, symbolic reasoning, and multi-step logic. Finally, the AIME dataset (Veeraboina, 2024) evaluates on competition-level problems from 1983–2024 that require creativity, deep conceptual understanding, and nontrivial problem decomposition. Evaluating on this range of datasets allows us to analyze how IIPC scales from straightforward arithmetic reasoning to advanced, competition-grade problem solving, and to identify the conditions under which iterative refinement yields the greatest benefit. Ideally, we aimed to evaluate approximately 1,500 problems per dataset to ensure balanced statistical representation. GSM8K only contains 1319 problems in its testing set, all of which were used in the ablation study. For MATH, while we aimed for uniform coverage across all topic–difficulty combinations (35 combinations total), certain combinations contained fewer available problems. This yielded a final sample of 1,483 problems instead of the intended 1,505 (the closest multiple of 35 to 1500). Finally, the AIME dataset contained a fixed set of 933 problems spanning all available years (1983–2024), and was used in its entirety.

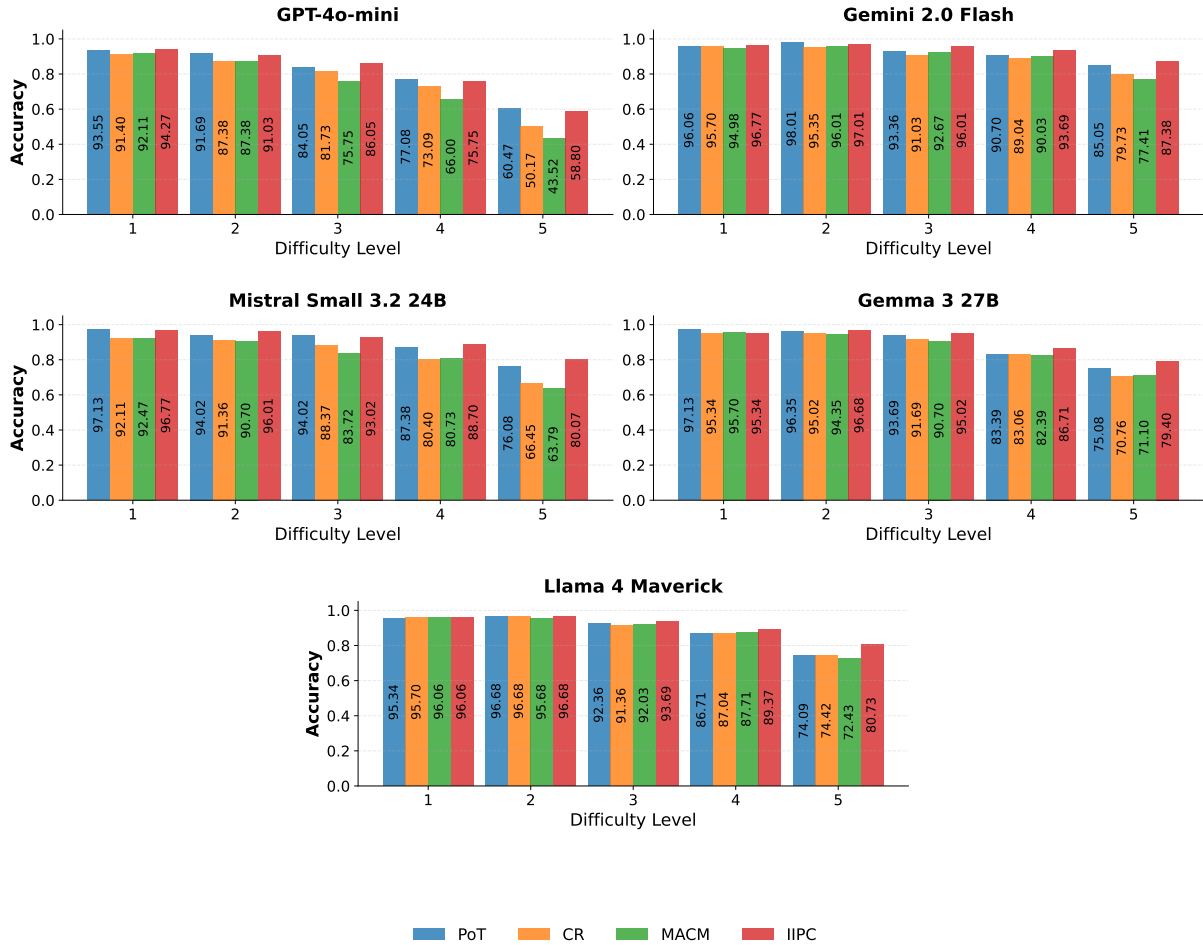


Figure 4: Accuracy by difficulty level for each LLM, comparing PoT, CR, MACM, and IIPC agents on the MATH dataset.

C Additional Experimental Results for MATH dataset

In our main experiments, We present a fine-grained analysis by problem difficulty and topic only for Llama 4 Maverick. This section extends this same per-difficulty and per-topic comparison to all other language models considered in this paper.

Across difficulty levels (Figure 4), IIPC demonstrates strong performance on Gemini 2.0 Flash, Mistral Small 3.2 24B, Gemma 3 27B, and Llama 4 Maverick, while slightly trailing PoT on GPT-4o mini. On Gemini 2.0 Flash, IIPC maintains high accuracy from 96.77% at Level 1 to 87.38% at Level 5, whereas MACM drops from 94.98% to 77.41%, a 9.97% difference at the highest difficulty. Similar trends are observed on Mistral 3.2 24B (IIPC: 96.77→80.07; MACM: 92.47→63.79), Gemma 3 27B (IIPC: 95.34→79.40; MACM: 95.70→71.10), and Llama 4 Maverick (IIPC: 96.06→80.73; MACM: 96.06→72.43), indicating that IIPC better preserves reasoning stability un-

der increasing problem complexity. In contrast, on GPT-4o-mini, PoT maintains an edge over IIPC. We also observe that IIPC maintains the highest accuracy on the most difficult problems on all models except GPT-4o-mini.

The topic-wise analysis (Figure 5) reveals similar patterns. On Gemini 2.0 Flash, IIPC leads in algebra (99.53%), number theory (98.51%), pre-algebra (97.67%), and precalculus (94.88%). On Gemma 3 27B, IIPC maintains a lead in geometry (82.38%), intermediate algebra (88.37%), prealgebra (94.88%), and precalculus (87.91%), outperforming other agents by multiple points. On Llama 4 Maverick, IIPC’s performance either matches or surpasses that of all other agents across most topics, demonstrating IIPC’s ability to fully utilize the reasoning capabilities of high performing models. On GPT-4o mini, PoT tends to lead IIPC by significant margins.

D IIPC Reasoning Examples

Figure ?? is a program refinement generated by Gemini 2.0 Flash when solving an AIME problem. IIPC allows the LLM to reflect on its previous program, identify the incorrect assumption, and propose the fix before developing a new program that produces the correct output.

Figure ?? demonstrates the fusion stage of the IIPC agent where it combines both reasoning branches to produce the final reasoning chain. Here, Llama 4 Maverick makes a mistake in its program by reversing the terms in the formula that calculated the dot product, resulting in a negative value. However, it identifies and applies the correct formula in the token-reasoning branch. In the final reasoning chain, which was conditioned on context from both branches, IIPC allows the LLM to deliberate between both sources and identify the error in the program branch, resulting in a correct final answer.

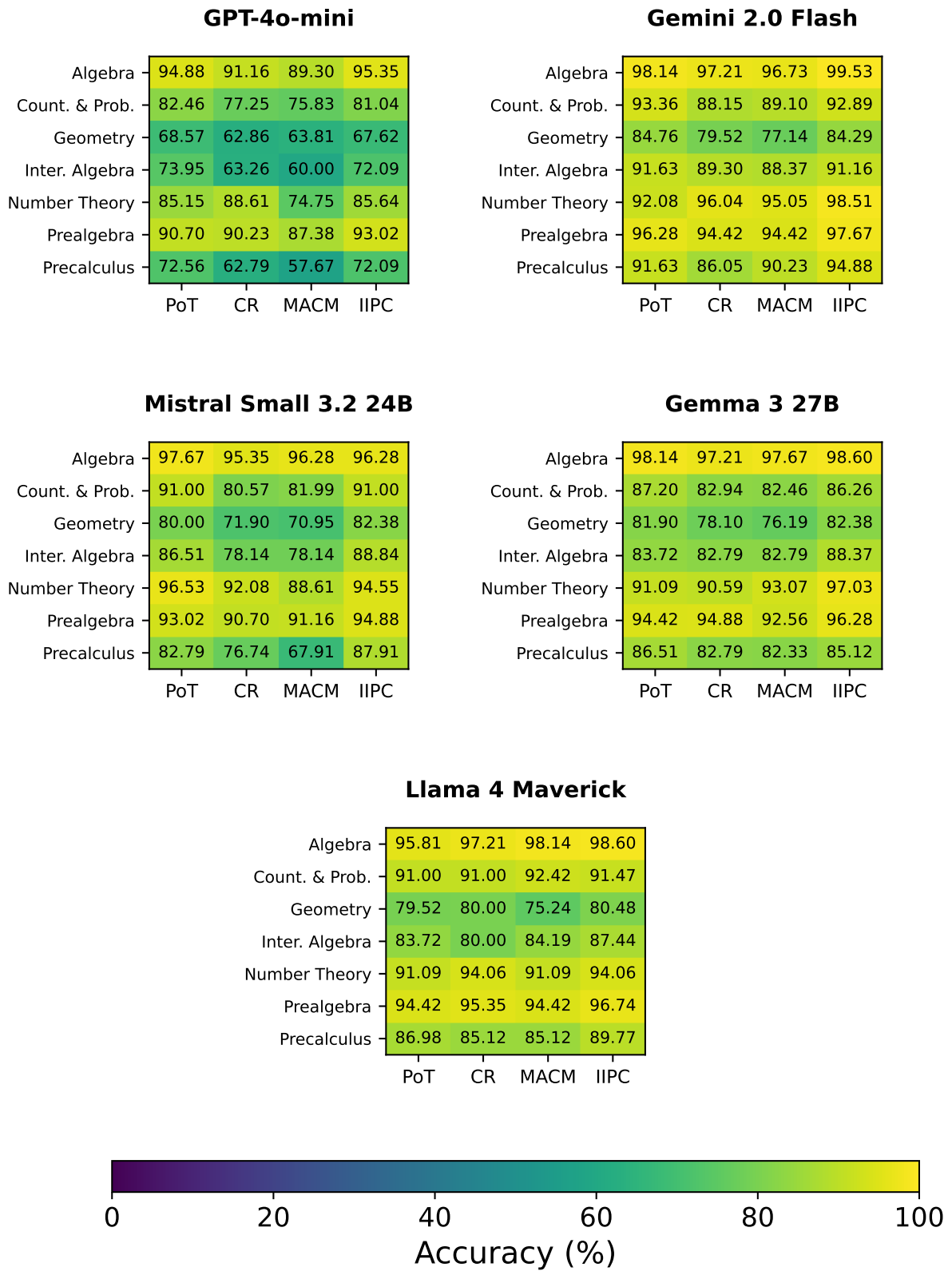


Figure 5: Heatmaps showing accuracy by topic for each LLM, comparing PoT, CR, MACM, and IIPC agents on the MATH dataset.

Problem: A right circular cone has a base with radius 600 and height $200\sqrt{7}$. A fly starts at a point on the surface of the cone whose distance from the vertex of the cone is 125, and crawls along the surface of the cone to a point on the exact opposite side of the cone whose distance from the vertex is $375\sqrt{2}$. Find the least distance that the fly could have crawled.

Original Program ✗:

```
python code ...  
  
# The angle difference is pi.  
d = math.sqrt(r1**2 + r2**2 - 2 * r1 * r2 * math.cos(math.pi))  
  
python code ...
```

Original Program Output ✗:

```
The slant height of the cone is: 800.0  
The central angle of the sector is: 4.71238898038469  
The shortest distance the fly could have crawled is:  
655.3300858899107  
655.3300858899107
```

Reflection: The previous program calculates the slant height and central angle correctly. However, it incorrectly assumes that "exact opposite side of the cone" translates to an angle difference of π in the flattened sector. The correct angle difference should be π radians *only if* the central angle of the sector is 2π . In general, the angle difference should be $\pi * (2\pi/\theta)$, where θ is the central angle of the sector. The program also lacks verification that the final answer is reasonable.

Refined Program ✓:

```
python code ...  
  
# Calculate the angle difference.  
delta_theta = math.pi * r / l  
print(f"The angle difference is: {delta_theta}")  
  
# Calculate the shortest distance using the law of cosines.  
d = math.sqrt(  
    r1**2 + r2**2 - 2 * r1 * r2 * math.cos(delta_theta)  
)  
  
python code ...
```

Refined Program Output ✓:

```
The slant height of the cone is: 800.0  
The central angle of the sector is: 4.71238898038469  
The angle difference is: 2.356194490192345  
The shortest distance the fly could have crawled is: 625.0  
The calculated distance is less than the sum of the distances  
from the vertex, which is expected.  
Performing alternative verification...  
Approximate shortest distance: 624.9134519586773
```

Figure 6: Example of IIPC's program refinement mechanism using Gemini 2.0 Flash. Only significant changes in the code are displayed to control figure size. The full example text is provided in the github repository

Problem: In triangle ABC , $AB = 3$ and $AC = 5$. Let O be the circumcenter of triangle ABC . Find $\vec{OA} \cdot \vec{BC}$.

Program Branch (Contains Flawed Logic X):

```
import numpy as np

# Define vectors AB and AC
AB = np.array([3, 0]) # Assuming AB lies on the x-axis

# Since we're working with a general triangle,
# let's consider the angle between AB and AC
# We'll directly use the property that  $OA \cdot BC = (AC^2 - AB^2)/2$ 

# Calculate  $OA \cdot BC$  using the formula derived from the
# properties of the circumcenter and dot product
AC_length = 5
AB_length = 3
OA_dot_BC = (AC_length**2 - AB_length**2) / 2

print("The dot product of OA and BC is: ", OA_dot_BC)

# Verification
assert np.isclose(
    OA_dot_BC,
    (5**2 - 3**2) / 2
), "Incorrect calculation"
```

Original Program Output X:

The dot product of OA and BC is: 8.0

CoT Branch Text ✓:

Step 1: To solve the problem, we need to start by understanding the given information and what is being asked. We are dealing with triangle ABC , where $AB = 3$ and $AC = 5$. The circumcenter of the triangle is O , and we are required to find the dot product of \vec{OA} and \vec{BC} .

Step 2: First, let's recall that the circumcenter O of a triangle is the point where the perpendicular bisectors of the sides of the triangle intersect. This means O is equidistant from the vertices A , B , and C .

...

Step 13: Now, we substitute $AB = 3$ and $AC = 5$ into the expression to find $\vec{OA} \cdot \vec{BC}$.

Step 14: $\vec{OA} \cdot \vec{BC} = \frac{3^2 - 5^2}{2} = \frac{9 - 25}{2} = \frac{-16}{2} = -8$.

The final answer is:

Final Reasoning Chain (Using Both Context Sources) ✓:

The objective of the given problem is to find the dot product of vectors \vec{OA} and \vec{BC} in triangle ABC , where $AB = 3$, $AC = 5$, and O is the circumcenter of the triangle.

Upon analyzing the context from both the code and the reasoning chain, it is evident that they both lead to the same formula for calculating $\vec{OA} \cdot \vec{BC}$, which is $\frac{AC^2 - AB^2}{2}$. The reasoning chain provides a detailed step-by-step derivation of this formula using properties of the circumcenter and dot product.

The code directly implements this formula to calculate $\vec{OA} \cdot \vec{BC}$. However, upon closer inspection, it is noticed that the code contains a logical error in its output. The correct calculation should yield a negative result because $AC > AB$, and the formula is $\frac{AB^2 - AC^2}{2}$.

To correct this, we should negate the output of the code or directly use the correct formula. The correct calculation is $\frac{3^2 - 5^2}{2} = \frac{9 - 25}{2} = -8$.

Combining the insights from both the code and the reasoning chain, we arrive at the final answer. The code is genuinely helpful in solving the problem and, with a correction, produces an accurate output.

The final answer is:

Figure 7: Example of IIPC's dual branch mechanism using Llama 4 Maverick. CoT output has been shortened to control image size. The full example text is provided in the github repository