Tales from a Graph: a Pipeline for Mathematical Problem Generation

Bastien Le Chenadec, Eleonora Kreacic, Mathieu Sibue, Gabriel Mercier[*], **Jannik Brinkmann**[*], **Nelson Vadori, Manuela Veloso**
J.P. Morgan AI Research

Abstract

We present a framework for generating pairs of mathematical and word problems with controllable complexity, where the shared underlying mathematical steps are guaranteed to be correct. Our approach enables controlled studies revealing systematic performance gaps between mathematical and word problem variants, while providing verifiable training signals for reasoning models as each problem pair is based on an underlying graph where each step is solved symbolically.

1 Introduction

OpenAI demonstrated the reasoning abilities of their `o1` model with the following problem [15]: *"A princess is as old as the prince will be when the princess is twice as old as the prince was when the princess's age was half the sum of their present ages. What are the ages of the prince and princess?"* At first glance the riddle appears intimidating, but its solution reduces to five linear equations.[1] This exemplifies a broader challenge in mathematical reasoning: while the underlying mathematics can be relatively simple, one difficulty lies in parsing the narrative and identifying relevant quantities to solve the problem. In figure 1 we illustrate that our pipeline is able to generate problems similar to the "prince and princess" example, starting from the mathematical equations only.

Math Problem

The quantities δ, τ, v, a, ε, b satisfy:

$$b = 7\delta + v$$
$$\tau = 5\delta$$
$$v = 3\delta + \tau$$
$$-\tau + \varepsilon = -\delta + a$$
$$b - v = -\tau + \varepsilon$$

The first value is known: $\delta = 4$.

(a) Using the relations above, determine v and b.
(b) Write the result as the vector $\mathbf{h} = (v, b)$.

Word Problem

In the seaside town of Lumenport, the lighthouse keeper records that at sunrise four brass lamps are lit on the rotating lantern. This count of four lamps will serve as the initial quantity for a series of related measurements.

According to the lighthouse maintenance log, the following relationships hold among the quantities:

- The total power consumption equals seven times the lamp count plus the electrical voltage.
- The rotation angle equals five times the lamp count.
- The electrical voltage equals three times the lamp count plus the rotation angle.
- The energy usage minus the rotation angle equals the illumination factor minus the lamp count.
- The total power consumption minus the electrical voltage equals the energy usage minus the rotation angle.

Using these relations and lamp count $= 4$, answer:
(a) Find the electrical voltage and the total power consumption.
(b) Record them as $(\text{voltage}, \text{power})$.

Figure 1: Problems generated by our pipeline. The underlying graph contains the 5 linear equations.

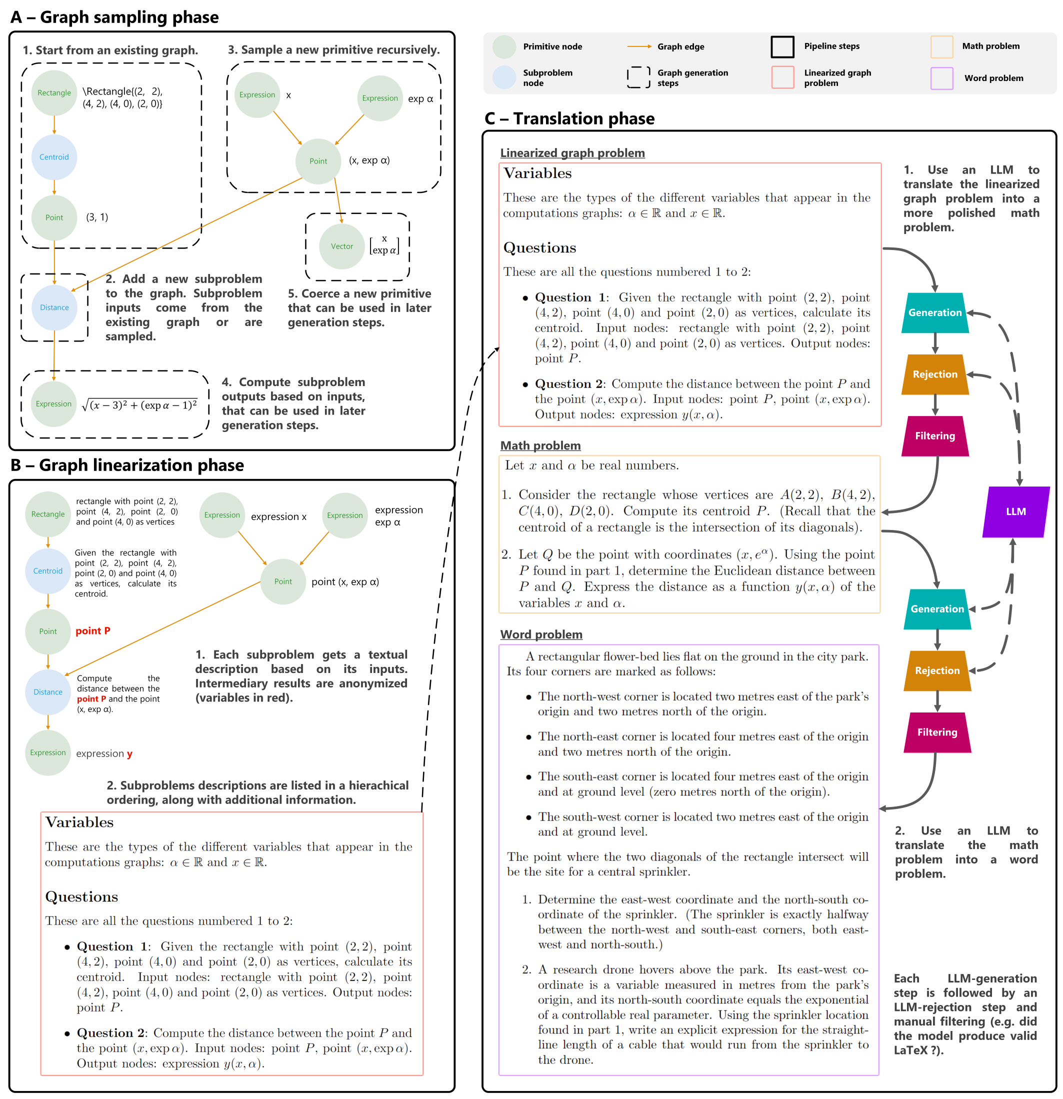
Contributions. We present a framework that generates paired math and word problems from mathematical objects such as equations, with controllable compositional complexity. Problems are represented as symbolic graphs, ensuring correctness of both intermediate steps and final answers. An LLM then translates these structures into textual math formulations and narrative word problems.

[*]Work completed in the context of an internship at J.P. Morgan AI Research.
[1]$a = y$, $b = 2z$, $2c = a + x$, $a - x = b - y = c - z$; x and a being the current prince and princess ages.

In contrast to prior work (see App. A), our framework uniquely combines SymPy-based symbolic computation for guaranteed correctness with a graph-sampling method that composes subproblems by input–output compatibility. An anonymization step further ensures that intermediate results are not leaked to the LLM during problem generation. Moreover, we present a scalable extension of subproblem coverage via LLM-assisted code generation.

2 Synthetic Generation of Math and Word Problems

We present a pipeline for synthetically generating complex mathematical problems with verifiable final and intermediate solutions (Figure 2). First, a computational graph is sampled using a heuristic algorithm. Then, the computational graph is linearized, i.e. a textual representation is generated for it. Finally, the textual representation gets translated into math and word problems using an LLM.


A – Graph sampling phase
1. Start from an existing graph.
Rectangle
\Rectangle{(2, 2), (4, 2), (4, 0), (2, 0)}
Centroid
Point
(3, 1)
3. Sample a new primitive recursively.
Expression
x
Expression
exp α
Point
(x, exp α)
Vector
$\begin{bmatrix} x \\ \exp\alpha \end{bmatrix}$
2. Add a new subproblem to the graph. Subproblem inputs come from the existing graph or are sampled.
Distance
5. Coerce a new primitive that can be used in later generation steps.
Expression
$\sqrt{(x-3)^2+(\exp\alpha-1)^2}$
4. Compute subproblem outputs based on inputs, that can be used in later generation steps.
Primitive node
Graph edge
Pipeline steps
Math problem
Subproblem node
Graph generation steps
Linearized graph problem
Word problem
B – Graph linearization phase
Rectangle
rectangle with point (2, 2), point (4, 2), point (2, 0) and point (4, 0) as vertices
Centroid
Given the rectangle with point (2, 2), point (4, 2), point (2, 0) and point (4, 0) as vertices, calculate its centroid.
Point
point P
Distance
Compute the distance between the point P and the point (x, exp α).
Expression
expression y
Expression
expression x
Expression
expression exp α
Point
point (x, exp α)
1. Each subproblem gets a textual description based on its inputs. Intermediary results are anonymized (variables in red).
2. Subproblems descriptions are listed in a hierachical ordering, along with additional information.
C – Translation phase
Linearized graph problem
1. Use an LLM to translate the linearized graph problem into a more polished math problem.
Generation
Rejection
Filtering
LLM
Math problem
Generation
Rejection
Filtering
Word problem
2. Use an LLM to translate the math problem into a word problem.
Each LLM-generation step is followed by an LLM-rejection step and manual filtering (e.g. did the model produce valid LaTeX ?).

Variables

These are the types of the different variables that appear in the computations graphs: $\alpha \in \mathbb{R}$ and $x \in \mathbb{R}$.

Questions

These are all the questions numbered 1 to 2:

- **Question 1**: Given the rectangle with point $(2,2)$, point $(4,2)$, point $(4,0)$ and point $(2,0)$ as vertices, calculate its centroid. Input nodes: rectangle with point $(2,2)$, point $(4,2)$, point $(4,0)$ and point $(2,0)$ as vertices. Output nodes: point P.
- **Question 2**: Compute the distance between the point P and the point $(x, \exp\alpha)$. Input nodes: point P, point $(x, \exp\alpha)$. Output nodes: expression $y(x,\alpha)$.

Variables

These are the types of the different variables that appear in the computations graphs: $\alpha \in \mathbb{R}$ and $x \in \mathbb{R}$.

Questions

These are all the questions numbered 1 to 2:

- **Question 1**: Given the rectangle with point $(2,2)$, point $(4,2)$, point $(4,0)$ and point $(2,0)$ as vertices, calculate its centroid. Input nodes: rectangle with point $(2,2)$, point $(4,2)$, point $(4,0)$ and point $(2,0)$ as vertices. Output nodes: point P.
- **Question 2**: Compute the distance between the point P and the point $(x, \exp\alpha)$. Input nodes: point P, point $(x, \exp\alpha)$. Output nodes: expression $y(x,\alpha)$.

Let x and α be real numbers.

1. Consider the rectangle whose vertices are $A(2,2)$, $B(4,2)$, $C(4,0)$, $D(2,0)$. Compute its centroid P. (Recall that the centroid of a rectangle is the intersection of its diagonals).
2. Let Q be the point with coordinates (x, e^{α}). Using the point P found in part 1, determine the Euclidean distance between P and Q. Express the distance as a function $y(x,\alpha)$ of the variables x and α.

A rectangular flower-bed lies flat on the ground in the city park. Its four corners are marked as follows:

- The north-west corner is located two metres east of the park's origin and two metres north of the origin.
- The north-east corner is located four metres east of the origin and two metres north of the origin.
- The south-east corner is located four metres east of the origin and at ground level (zero metres north of the origin).
- The south-west corner is located two metres east of the origin and at ground level.

The point where the two diagonals of the rectangle intersect will be the site for a central sprinkler.

1. Determine the east-west coordinate and the north-south coordinate of the sprinkler. (The sprinkler is exactly halfway between the north-west and south-east corners, both east-west and north-south.)
2. A research drone hovers above the park. Its east-west coordinate is a variable measured in metres from the park's origin, and its north-south coordinate equals the exponential of a controllable real parameter. Using the sprinkler location found in part 1, write an explicit expression for the straight-line length of a cable that would run from the sprinkler to the drone.

Figure 2: Overview of the pipeline. (A): new graphs are sampled with a heuristic approach. (B): graphs are automatically linearized to be fed to an LLM. (C): an LLM is used to translate the linearized graphs to math problems and then word problems, including anonymization of intermediary results.

Sampling and linearizing computational graphs. We aim at sampling computational graphs combining multiple subproblems into a more complex problem. In contrast to similar approaches [21] [20], we distinguish between different *data modalities* (e.g. real functions, geometric objects, etc.) using the notion of *primitive*. Each kind of *subproblem* (e.g., integration, calculating an area, etc.) defines input and output interfaces in terms of existing primitives. We argue that this approach allows for better combinations of subproblems from different mathematical domains.

To facilitate mathematical manipulation of objects, we enforce that all primitives, at their core, should be objects from the `SymPy` python library (though our approach would allow for other types of objects). A *primitive* is characterized by i) a sampling method which can either be direct or recursive (i.e. builds on other primitives, see Figure 2 A-3.). ii) a textual representation method for the primitive, which is left to the `SymPy` printer configured to output LaTeX, but can also be overridden. A *subproblem* is characterized by i) the interface, i.e. its input and output primitives.[2] ii) the sampling method that computes the outputs based on the inputs. iii) a textual representation for the subproblem, which relies on the input primitives representations (Figure 2 B). While these elements enable sampling interesting and challenging problems, they can also easily be generated by an LLM to further automate the creation of new problems (Section 3).

Graph Generation. We sample new graphs by iteratively choosing new subproblems to be added, finding correct inputs based on the subproblem interface (either sampling new inputs or taking existing ones from the graph), and finally adding subproblem outputs to the graph. Generation is stopped when enough subproblems have been generated. Because subproblems can have multiple inputs and multiple outputs, the final problem is not constrained to be a linear succession of questions, but can display more complex dependencies.

Linearization. Turning the computational graph into a textual description relies on the linearization methods defined at the primitive and subproblem level. We pick an ordering to facilitate the translation (e.g. successive subproblems are next to each other). Finally, in order to keep the solutions to the intermediary subproblems hidden, we *anonymize* intermediary primitives with variable names.

Math and Word Problem Generation. The content of the linearized graph is translated into a single *Math Problem* using an LLM. The task only involves reformulating and rearranging information, as intermediary results that would need to be hidden are already anonymized in the graph linearization step. Translating the math problem to a word problem is more complex, because the model has to design a coherent scenario related to the same underlying mathematical content. To tackle this problem, we use *step-back prompting* [27], first instructing the model to find coherent themes for each mathematical object appearing in the math problem, and then generating the whole narrative connecting everything together. Both the math and word problem creation steps are followed by a rejection step where an LLM decides whether the problem is well-formed, and a filtering step based on whether the generation constitutes valid LaTeX. More details are available in Appendix D.

3 Automatic Subproblem Creation Using LLMs for Code Generation

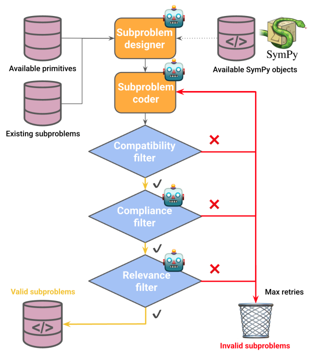
Our pipeline relies on pre-defined primitive and subproblem classes to construct computational graphs, requiring manual implementation for new types or operations. To reduce this burden and demonstrate scalability, we explore automating subproblem code generation with LLMs. More details are presented in Appendix C.

Subproblem Code Desiderata. We require new subproblems to satisfy: (i) *code compatibility*, (ii) *instruction compliance*, (iii) *mathematical relevance*. **Code compatibility** checks adherence to our codebase's design principles. **Instruction compliance** ensures the code faithfully implements the intended mathematical idea. **Mathematical relevance** demands non-trivial and sound subproblems.

Methodology. To meet these quality criteria, we use an agentic workflow that separates subproblem ideation from code implementation: an LLM first designs subproblem specifications given a mathematical theme [26], which a coder LLM then implements (Figure 3). The latter iteratively refines its output based on targeted feedback from filters until all conditions are met or a maximum number of retries is reached. Code compatibility is tested by instantiating the class and trying to attach it to random input primitive nodes. Instruction compliance and mathematical relevance are assessed by LLM judges, due to the challenge of programmatically verifying these.

Synthetic Subproblem Statistics. We evaluate how backbone LLM choice, temperature, reasoning effort, and prompting method for the subproblem designer influence key statistics of the generated

[2]Order of inputs & outputs matter, so the overall graphs we generate are DAG with partial order on vertices.



(a) Subproblem generation setup

```
class AngleBetweenVectorsProblem(Problem[AngleBetweenVectorsProblemConfig],
↪   Inputs[VectorPrimitive, VectorPrimitive], Outputs[ExprPrimitive]):

    def create_problem(self, vec1: VectorPrimitive, vec2: VectorPrimitive) ->
    ↪   Tuple[ExprPrimitive]:
        v1 = vec1.content
        v2 = vec2.content
        if v1.shape[0] != v1.shape[1] and v1.shape[1] != 1:
            raise NotImplementedError("First input is not a vector.")
        if v2.shape[0] != v2.shape[1] and v2.shape[1] != 1:
            raise NotImplementedError("Second input is not a vector.")
        norm1 = sp.sqrt(v1.dot(v1))
        norm2 = sp.sqrt(v2.dot(v2))
        if norm1 == 0 or norm2 == 0:
            raise NotImplementedError("Angle is undefined because one of the vectors is
            ↪   zero.")
        cos_theta = v1.dot(v2) / (norm1 * norm2)
        angle = sp.acos(cos_theta)
        return (ExprPrimitive(content=angle),)

    def core_description(self, vec1: VectorPrimitive, vec2: VectorPrimitive) -> str:
        return (f"Compute the angle between the {vec1.description()} and the
        ↪   {vec2.description()} (in radians).")
```

(b) Example subproblem class generated using `gpt-oss-120b`

Figure 3: Overview of automated subproblem code generation using LLMs.

subproblems. As shown in Table 1, LLMs can cheaply expand our subproblem set with novel, high-quality instances. We observe that larger models like `gpt-oss-120b` produce more valid and slightly more varied outputs than smaller ones. As expected, a higher T value for the subproblem designer LLM increases the use of unique `SymPy` objects, but also raises the rate of invalid problems. A greater R reduces the variety of `SymPy` objects but enhances diversity in subproblem descriptions, indicating a potentially more creative use of fewer objects. Finally, supplying theme-relevant `SymPy` objects in the prompt significantly boosts subproblem diversity by guiding idea generation beyond the internal knowledge of the subproblem designer LLM. While synthetic subproblems do not perfectly match manual ones in all metrics, the approach is highly promising –especially as some metrics naturally favor manual data (e.g., primitive usage, since certain primitives were specifically added for those subproblems).

Table 1: Comparison of generation strategies for 100 subproblems & 10 max. retries, over 3 seeds. *("uninst."=uninstantiable, "Compat."=compatibility, "Compli."=compliance, "Relev."=relevance.)*

Subproblem design & code strategy	**Valid sub--problems (↑)**	**Debugging iterations (↓)**	**Class uninst. (↓)**	**Compat. fail (↓)**	**Compli. fail (↓)**	**Relev. fail (↓)**	**Test passes (↑)**	**Primitives used (↑)**	**Unique SymPy objects (↑)**	**SymPy objects rarity (↑)**	**Description embedding similarity (↓)**
Manual	42/42	–	–	–	–	–	78.1%	86.4%	57.0	0.64	0.146
`Devstral-Small-2507`	32.0/100	7.5	1.0%	71.0%	27.0%	**1.0%**	**89.4%**	72.7%	41.7	**0.55**	0.188
`gpt-oss-20b`	78.0/100	4.7	1.2%	65.1%	18.1%	11.6%	87.6%	65.2%	54.2	0.45	0.201
`gpt-oss-120b`	**95.3/100**	2.5	**0%**	**6.7%**	57.8%	35.5%	87.4%	74.2%	53.4	0.45	0.188
w/ $R =$ high	92.6/100	**2.2**	4.2%	26.1%	**12.0%**	57.7%	89.3%	71.2%	48.0	0.41	**0.170**
w/ $T = 1$	91.0/100	2.7	**0%**	23.9%	33.9%	42.2%	88.7%	71.2%	56.2	0.46	0.191
w/ SymPy prompt	93.0/100	2.7	**0%**	20.6%	38.7%	40.7%	86.2%	**80.3%**	**87.0**	0.53	0.179

4 Effects of Compositional Depth and Problem Formulation

We use our generation pipeline to evaluate models along two controlled axes: (i) *graph size*, which controls the compositional depth of the problems, and (ii) *problem form*, comparing math and word problems. These experiments serve as case studies for the type of targeted analyses that the pipeline enables. We use `gpt-oss-120b` within our pipeline to generate the datasets, and adopt an LLM-as-a-judge setup to verify whether a predicted solution matches the ground-truth answer. We noticed better performance of the latter compared to a Math-Verify-based parser.

Performance degrades systematically with graph size. Figure 4 (left) shows accuracy against graph size (2-6 nodes) for five models. Consistently, models exhibit sharp performance drops as compositional depth increases. The decline is steepest beyond single-hop dependencies, where models must track intermediate results across steps. For example, `Llama-3.2-3B` drops from 32% to 15% accuracy, while `gpt-oss-120B` falls from 73% to 46%. Notably, even `gpt-oss` reasoning models show similar degradation patterns as standard language models, indicating that multi-step symbolic composition challenges persist despite specialized training.

Word problems create extra difficulties beyond just symbolic reasoning. Figure 4 (right) shows that word problems are consistently harder than math problems with identical structure (graph size

= 1). For example, `Llama-3.2-3B` achieves around 45% on math versus 35% on word problems. For `gpt-oss` models, the difference is notably larger: `gpt-oss-20b` and `120b` reach nearly 80% accuracy on math but their accuracy drops by 25-30% on word problems. This pattern reveals that applying the same reasoning within a narrative context poses significant additional challenges.

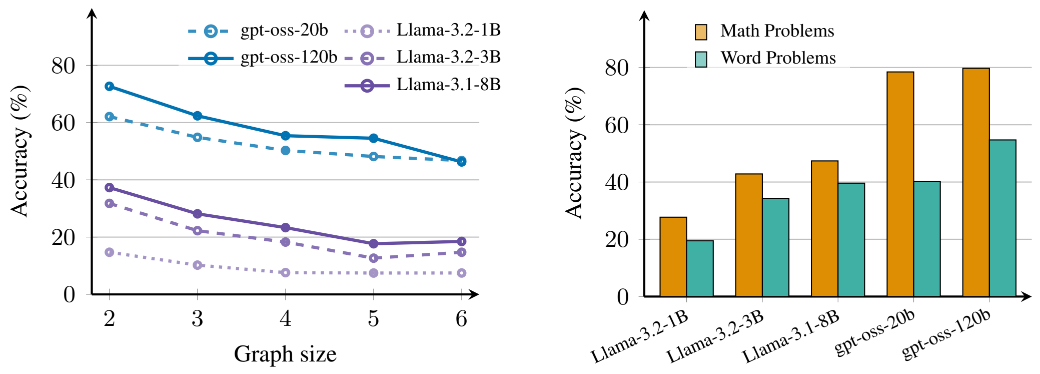
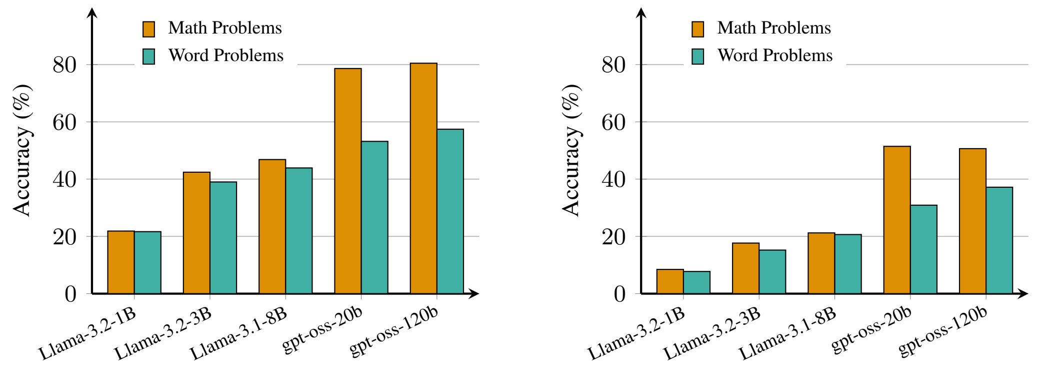


Figure 4: **(left)** Performance declines systematically with graph size **(right)** Comparing performance on math and word problems shows that some models consistently struggle with word problems.

Medium and hard problems. The results in Figure 4 were computed on problems with an easy-configuration, i.e. the expressions and subproblem hyper-parameters were rather simple. In figure 5 we present additional results on problem with medium and hard configurations. Harder hyper-parameters result in steep performance drops across all evaluated models.



(a) Medium Problem Complexity

(b) Hard Problem Complexity

Figure 5: Accuracy declines for increasing problem complexity, despite keeping the size of the graph fixed (graph size = 1).

5 Conclusion and Future Work

In this work, we construct mathematical problems by combining a series of subproblems, each of which can be solved using `SymPy`. These subproblems form a graph, where two subproblems are connected to each other only if they have compatible inputs / outputs. This graph information is then fed to a LLM in charge of formulating both a mathematical problem and a word problem corresponding to the same underlying mathematical content. We implement a series of LLM-based filters to ensure that the created problems are coherent with the graph, and show how the method can be scaled by using LLMs to build code for new subproblems that can be incorporated in the graph. Our experiments reveal systematic performance gaps between mathematical and word problem variants, as well as show that problems of various complexity still represent a challenge for some recent open-source models. In a future work, we plan on using this framework for model training.

Disclaimer

References

[1] M. Balunovic, J. Dekoninck, I. Petrov, N. Jovanovic, and M. T. Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *CoRR*, abs/2505.23281, May 2025.

[2] Z. Chen, Y. Chen, J. Han, Z. Huang, J. Qi, and Y. Zhou. An empirical study of data ability boundary in llms' math reasoning, 2024.

[3] K. Cobbe, V. Kosaraju, M. Bavarian, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. Training verifiers to solve math word problems, 2021.

[4] N. Dziri, X. Lu, M. Sclar, X. L. Li, L. Jiang, B. Y. Lin, S. Welleck, P. West, C. Bhagavatula, R. L. Bras, J. D. Hwang, S. Sanyal, X. Ren, A. Ettinger, Z. Harchaoui, and Y. Choi. Faith and fate: Limits of transformers on compositionality. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[5] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

[6] K. Hong, A. Troynikov, and J. Huber. Context rot: How increasing input tokens impacts llm performance. `https://research.trychroma.com/context-rot`, July 2025.

[7] C. Li, Z. Yuan, H. Yuan, G. Dong, K. Lu, J. Wu, C. Tan, X. Wang, and C. Zhou. Mugglemath: Assessing the impact of query and response augmentation on math reasoning, 2024.

[8] C. Li, Z. Yuan, H. Yuan, G. Dong, K. Lu, J. Wu, C. Tan, X. Wang, and C. Zhou. Query and response augmentation cannot help out-of-domain math reasoning generalization, 2024.

[9] J. LI, E. Beeching, L. Tunstall, B. Lipkin, R. Soletskyi, S. C. Huang, K. Rasul, L. Yu, A. Jiang, Z. Shen, Z. Qin, B. Dong, L. Zhou, Y. Fleureau, G. Lample, and S. Polu. Numina-math. `[https://huggingface.co/AI-MO/NuminaMath-CoT](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf)`, 2024.

[10] Z. Li, Z. Zhou, Y. Yao, X. Zhang, Y.-F. Li, C. Cao, F. Yang, and X. Ma. Neuro-symbolic data generation for math reasoning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[11] H. Luo, Q. Sun, C. Xu, P. Zhao, J.-G. Lou, C. Tao, X. Geng, Q. Lin, S. Chen, Y. Tang, and D. Zhang. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. In *The Thirteenth International Conference on Learning Representations*, 2025.

[12] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *https://arxiv.org/pdf/2410.05229*, 2024.

[13] S. I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar. GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

[14] S. Mishra, G. Poesia, and N. Goodman. From Next-Token to Mathematics: The Learning Dynamics of Mathematical Reasoning in Language Models. In *COLM, the Conference On Language Modeling*, 2025.

[15] OpenAI. Logic Puzzles with OpenAI o1. *https://www.youtube.com/watch?v=Jh2NdbPDVrQ*, 2024.

[16] I. Petrov, J. Dekoninck, L. Baltadzhiev, M. Drencheva, K. Minchev, M. Balunovic, N. Jovanović, and M. Vechev. Proof or bluff? evaluating LLMs on 2025 USA math olympiad. In *2nd AI for Math Workshop @ ICML 2025*, 2025.

[17] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, and D. Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *https://arxiv.org/pdf/2402.03300*, 2024.

[18] P. Shojaee, I. Mirzadeh, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, 2025.

[19] Z. Tang, X. Zhang, B. Wang, and F. Wei. Mathscale: Scaling instruction tuning for mathematical reasoning, 2024.

[20] J. Wang, J. Jiang, Z. Zhang, J. Zhou, and W. X. Zhao. Rv-syn: Rational and verifiable mathematical reasoning data synthesis based on structured function library, 2025.

[21] J. Wang, J. Xu, X. Wang, Y. Wang, M. Xing, S. Fang, Z. Chen, H. Xie, and Y. Zhang. A graph-based synthetic data pipeline for scaling high-quality reasoning instructions, 2025.

[22] K. Wang, H. Ren, A. Zhou, Z. Lu, S. Luo, W. Shi, R. Zhang, L. Song, M. Zhan, and H. Li. Mathcoder: Seamless code integration in LLMs for enhanced mathematical reasoning. In *The Twelfth International Conference on Learning Representations*, 2024.

[23] P. Wang, L. Li, Z. Shao, R. X. Xu, D. Dai, Y. Li, D. Chen, Y. Wu, and Z. Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations, 2024.

[24] T. Ye, Z. Xu, Y. Li, and Z. Allen-Zhu. Physics of Language Models: Part 2.1, Grade-School Math and the Hidden Reasoning Process. *ICLR*, 2025.

[25] L. Yu, W. Jiang, H. Shi, J. YU, Z. Liu, Y. Zhang, J. Kwok, Z. Li, A. Weller, and W. Liu. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[26] Y. Yu, Y. Zhuang, J. Zhang, Y. Meng, A. J. Ratner, R. Krishna, J. Shen, and C. Zhang. Large language model as attributed training data generator: A tale of diversity and bias. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[27] H. S. Zheng, S. Mishra, X. Chen, H.-T. Cheng, E. H. Chi, Q. V. Le, and D. Zhou. Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models. *ICLR*, 2024.

[28] K. Zhu, J. Chen, J. Wang, N. Z. Gong, D. Yang, and X. Xie. Dyval: Dynamic evaluation of large language models for reasoning tasks. In *The Twelfth International Conference on Learning Representations*, 2024.

A Related Work

Similarly to us, MathCAMPS [14] uses `SymPy` to get answers to the symbolic problems and an LLM to translate them into word problems. Whilst our pipeline enables the integration of subproblems from multiple math domains into a single problem, their approach is limited to problems defined by school curriculum standards and does not examine how distinct standards can be combined in a graph-like manner to form more complex problems. GRIP [21] build a graph of relationships between mathematical concepts (e.g., theorems) based on their co-occurrence in the same seed problem, and then employ prompting in order to generate math problems that combine related concepts. Unlike our method that generates problem solutions with `SymPy`, GRIP employs LLMs to generate solutions and thus has no correctness guarantee. The most directly comparable approach to ours is RV-Syn [20]. They build a graph of relations between simpler math functionalities (analogous to our subproblems) extracted from seed data, and sample computational graphs that combine these functionalities. Finally, they utilize LLMs to generate math problems from those computational graphs that passed validity checks based on successful Python execution. Our approach in contrast does not model relationships between subproblems but combines them solely based on type compatibility of the inputs and outputs, which promotes cross-domain integration and enables scaling up our approach through automatic generation of new subproblems.

Mathematical Reasoning. Performance of LLMs on mathematical reasoning tasks has served as a proxy of their general intelligence capabilities. Although LLMs have recently achieved excellent results on math Olympiad-level problems, the question of whether they are truly capable of math reasoning remains open. Firstly, it is difficult to accurately quantify the progress because existing benchmarks carry high risk of data contamination in training [1, 16]. Moreover, due to inherently compositional nature of math problems it is necessary to understand performance of LLMs on a wide range of problems that exhibit various levels of complexity, instead of relying on fixed datasets [4, 18, 12]. Thus it is highly desirable to provide methods that enable flexible generation, yet at the same time guarantee correctness of the generated math reasoning data. In addition to being used as benchmarks in evaluating LLMs, math reasoning datasets also help improve their reasoning capabilities. Historically there has been a wide gap in math reasoning performance between proprietary and open source models, however open-source models are rapidly improving and narrowing the gap. A common approach for improving abilities of open source models involves some form of data augmentation followed by supervised fine-tuning (SFT) on the new dataset that consists of pairs of math problems and detailed reasoning paths [2, 25, 7]. After the SFT stage, [11] and [23] further rely on RL with process reward model that assigns a score to each reasoning step. DeepSeekMath [17] introduced Group Relative Policy Optimization (GRPO) with accuracy reward approach to learn from pairs consisting of a problem and a *final answer* only, thus removing the need for full reasoning traces. In [24], they design a series of experiments to analyze the reasoning process of LLMs, in particular they build synthetic GSM8K-like datasets in order to capture a variety of dependencies between parameters. There, the goal is not to study arbitrary systems of linear equations among parameters but rather to compute parameters one after the other using simple operations.

Benchmarks. Prominent math reasoning benchmarks GSM8K [3] and MATH [5], consist of grade school math word problems and high-school math competitions problems, respectively. AIME problems (created for a specific competition year) are extremely challenging problems that serve as one of criteria for IMO qualification. As there is a significant gap in the level of problem difficulty represented by such benchmarks, a number of works create new datasets by combining or augmenting existing ones (e.g. MwpBench [19], NuminaMath-CoT [9]). Such approaches yield new data with high degree of similarity with the seed data, that thus inherits aforementioned data contamination vulnerability. Moreover, close similarity translates to generating problems of similar complexity. However in order to gain detailed understanding of LLMs reasoning abilities, it is desirable to test them on a number of instances of the same problem with varying degrees of problem complexity [18]. Recent efforts to create datasets that vary in math problem nature and complexity, can be broadly categorized into prompt-based, symbolic and hybrid methods.

Prompt-based Methods. MetaMath [25] and MuggleMath [7] rely on prompt engineering in order to augment seed data by introducing simple modifications (e.g. rephrase questions, change specific numbers etc.) Although fine-tuning LLMs on augmented datasets improves their reasoning performance on in-domain math problems, it typically does not yield generalization to out-of-domain problems [8]. [19] develop a refined prompting strategy MathScale in order to extract mathematical concepts and their relationships from the seed data, which are then used as input to prompt templates for generation of new problems. Whilst they improve diversity, methods that rely solely on prompting typically lack control of mathematical rigor and are thus prone to compromising mathematical coherence of the generated problems.

Symbolic Methods. GSM-Symbolic [13] convert GSM8K problems into symbolic templates, by identifying variables within the problems, defining their ranges and establishing conditions between them to ensure mathematical correctness. Once created, symbolic templates are verified through automated checks, and subsequently used in order to generate new problems by sampling new values for the variables. DyVal [28] generate problems by employing: 1) sample generation algorithm based on directed acyclic graphs (DAGs), 2) the set of constraints for the generation process, to ensure problem correctness and desired level of complexity, and 3) description function to translate generated samples into natural language. Diversity of the generated samples comes from random sampling of values and operations, however this is limited by the constraints on permissible ranges/operations. Moreover, translation into natural language is achieved via a task dependent function, which does not capture diverse ways humans phrase math problems.

Hybrid Methods. Ours belongs to the family of hybrid methods that combine math coherence of symbolic approaches with creativity of prompt-based ones. Contrary to our approach which generates problems out of thin air, a number of works in this corpus relies on the seed data. [22] propose a method for generating novel math problems and the corresponding solutions based on the seed data consisting of math problems, natural language reasoning, python code, and its output. New data is obtained by prompting an LLM with a pair consisting of a simple and a difficult problem, in order to generate a problem of intermediary complexity. [10] formalize problems in the seed dataset in SMT-LIB language. Formalized problems then get mutated (complicated or simplified) into new symbolic representations which are by construction guaranteed to be mathematically correct, and finally with the use of an LLM they get translated into the new set of natural language problems. The limited set of mutation operators however poses limitation is generating more complex problems. Our pipeline on the other hand enables flexible composition of subproblems into more complex ones. In contrast to the methods relying to the seed data, [14] encode each skill prescribed by the school level mathematics curriculum in a formal grammar, which allows sampling a number of symbolically represented math problems targeting a certain skill. Their pipeline employs LLM to translate problems into the word problems, however as there is no guarantee that the word problem faithfully represents the original symbolic problem, they instruct LLM to translate the new word problem into its symbolic representation. They rely on symbolic solver `SymPy` to get the answers from symbolic representations which are then utilized to check the consistency between the original symbolic representations and those corresponding to the new word problems. Whilst our pipeline also utilizes `SymPy` to obtain answers to symbolically represented problems and an LLM to translate those into their word problem counterparts, the scope of problems considered by [14] is bounded to a school standard of math curriculum. In contrast to ours, their approach promotes diversity solely by sampling random values for the variables figuring in the particular curriculum standard.

Graph-based Problem Synthesis. A number of novel methods (including ours) represent the inner logic of math problems by graphs. MathScale [19] and GSDP [21] rely on seed data in order to extract relationships between mathematical concepts and build graphs of knowledge points (KP) which are further utilized to sample new problems. With the use of prompting, MathScale first extracts two kinds of high level concepts: topics (math areas, e.g. arithmetic) and knowledge points (KPs, e.g. specific theorems). They then proceed by constructing a graph with three types of edges (topic-topic, topic-KP, and KP-KP) effectively creating three subgraphs, where weight of each edge corresponds to the co-occurrence of the corresponding nodes in the seed data. Then, they sample concept composition by employing graph random walk on the created subgraphs, starting from the topic subgraph then moving to the topic-KP subgraph and finally finishing at the KP-KP subgraph. The obtained set of sampled topics and KPs is provided to LLM to generate problem/answers pairs. Similarly, GSDP constructs a graph of KP relationships, where each KP corresponds to a precise mathematical concept (e.g. a theorem or an algorithm) and an edge indicates that two KPs are found in the same seed problem. Prompting in combination to careful utilization of the graph topology yields the generation of the diverse data of problems. Namely, the graph is utilized not only through direct edge relationships: 2-hop and 3-hop relationships (consisting of all pairs of knowledge points with the shortest path of length 2 and 3, respectively) are also candidates for problem generation. In these approaches however there is no guarantee of mathematical correctness of the generated problems and corresponding solutions. RV-Syn [20] employ a different approach and for a set of seed data of mathematical problems prompt the model to generate computational graphs of mathematical functions that capture the flow of the solution processes. These are then used to build a function library in graph format where each node represents a function and edges relations between them, which further enables sampling new computational graphs that can be translated to new mathematical problems. Our approach in contrast does not rely on the seed data to extract relationships between subproblems, but flexibly combines them as long as there is compatibility between relevant inputs and outputs. Aforementioned DyVal [28] compose more complex problems as DAGs from fundamental elements represented by nodes. For problems of hierarchical nature (e.g. arithmetic or logic), leaf nodes correspond to starting points of the problem and their values are randomly selected, whilst for the rest of nodes the operation that defines relationship between the node and its children is randomly selected. For problems that are represented by general DAGs (i.e. not trees), the randomness comes from sampling nodes' values and their children. Each DAG is built to respect task constraints (permissible range of values, allowed operations) and complexity constraints (e.g. depth and width). Although their method achieves certain degree of diversity by allowing for various configurations of random values and operations between them, this is limited by the constraints on value ranges and set of allowed operations. On the other hand, in our pipeline each graph represents a math problem composed of a number of sub-problems and the compatible input/output primitives thus allowing for greater diversity of the resulting problems. Moreover, DyVal translates graphs to natural language with the use of a task specific function, whilst we prompt in order to translate our graphs into math problem and then subsequently its word problem counterpart which further promotes novelty.

B Additional Experiments

Per-problem results. In Figure 4, we compared model performance on the same problem with different formulations (math vs. word). Here we present additional results, A closer look at the per-problem performance of models on different problem formations reveals a more nuanced picture. For example, while gpt-oss-120b on average performs significantly worse on word formulations than on their math counterparts, it performs better on word formulations for some problems, e.g. for BinomialProbNumSuccessesProblem (see Figure 6). Interestingly, we observe that models perform systematically better on word formulations than on math formulations for problems that involve probabilities.

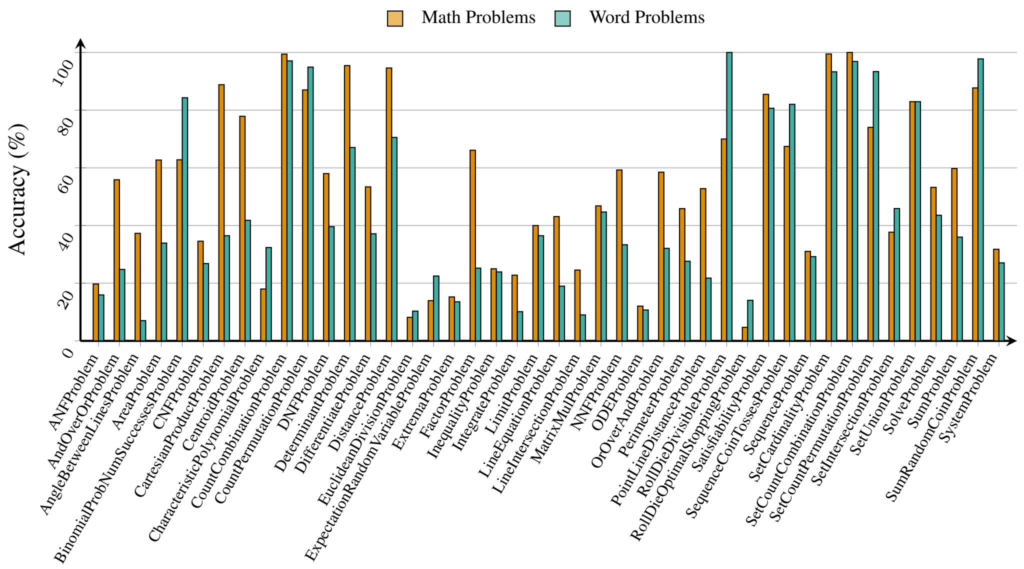


Figure 6: While overall performance is higher on math formulations, the model shows varied behavior across problem types. Notably, it performs better on word formulations for probability-related problems.

C Automated Subproblem Generation Details

Manually adding new primitive and subproblem classes is feasible for small numbers, but does not scale well. To address this, we propose a method to **automate the generation of new subproblems**, leveraging recent LLMs' strong coding capabilities to implement new subproblem classes in Python.

C.1 Subproblem Code Desiderata

Before generating new subproblems with LLMs, we establish formal criteria for subproblem suitability in our framework. We outline three key pillars:

- **Code compatibility:** The subproblem class must correctly execute within the graph sampling pipeline and conform to our interface conventions.
- **Instruction compliance:** The subproblem's behavior should align with the intent indicated by its name (and formulated in the subproblem code generation instructions). For example, the class `InverseMatrixProblem` should generate subproblem nodes involving matrix inversion.
- **Mathematical Relevance:** The problem should be non-trivial, sound, and ideally involve interactions between multiple primitives (thus increasing connectivity in the problem graph).

C.2 Methodology

We break down the complex task of implementing valid subproblem classes into simpler individual steps executed by separate LLM agents, as illustrated in Figure 3.

Subproblem Designer. A dedicated agent first designs novel subproblem specifications one after the other, given a uniformly sampled mathematical theme amongst {Geometry, Calculus, Algebra, Linear Algebra, Probability} [26]. We also add a free-choice theme for greater diversity. The prompt of the subproblem designer is structured as follows:

- It first describes the overall logic of our graph-sampling pipeline: every subproblem node connects into the problem graph using input and output primitives. A subproblem class should produce nodes that successfully integrate into computational graphs at least k out of N times given random input nodes. Each subproblem class should tackle a mathematical task whose resolution requires reasoning and preferably involves a single `SymPy` operation. Finally, every random input to a node produced by the subproblem class should correspond to a unique correct output (which can be multiple nodes, as long as this solution is unique).
- It enumerates the list of available primitives as well as the list of existing subproblems (iteratively updated), so the subproblem designer LLM avoids duplicating subproblem ideas and does not make up primitives.
- It states the mathematical thematic focus we want for each new subproblem.
- It specifies the expected output format. The subproblem designer LLM must return a single Markdown row describing the specifications of the prospective subproblem:
`| Name | Input (Primitives) | Output (Primitives) | Description |`.
The wording in the `description` must be precise and unambiguous.
- It provides five few shot examples of (math theme, subproblem specifications) pairs.
- Optionally, we supply the subproblem designer LLM with a list of theme-relevant `SymPy` objects (such as functions, classes, and variables) automatically extracted from the `SymPy` library. This serves as an external source of inspiration for generating new subproblems (see `SymPy` prompt results in Table 1).

Each subproblem specification returned by the problem designer is then used in a subsequent code implementation phase handled by other agents.

Subproblem Coder. A separate LLM coding agent turns the textual specifications from the LLM designer into a Python module that should be fully compatible with our graph pipeline. The guidelines provided to the subproblem coder for implementing an adequate Python module are the following:

- The output code should be wrapped between ```` ```python ```` and ```` ``` ````, and start with a comment header repeating the specifications of the problem idea (`Name`, `Input`, `Output`, `Description`), followed by the mandatory imports from `SymPy` and our internal framework.
- The output code should define a subproblem class named `<SubProblemName>Problem`, which should inherit from `Problem[<Config>]`, `Inputs[<Primitive1>,...]`, and `Outputs[<Primitive2>,...]`. It should implement two methods:
  - `create_problem` to apply `SymPy` operations to the input primitive(s), and return one (or a tuple of) output primitive(s), while handling edge cases via try/except statements that raise exceptions;
  - `core_description` to produce a valid LaTeXstring that describes the mathematical subproblem tackled, using the `description method` of the input primitive nodes.
- The output code should define a `dataclass <SubProblemName>ProblemConfig` to hold tunable parameters that the subproblem class could reuse (e.g. ranges for random sampling).
- It should ensure that the task to solve in the subproblem class admits a unique solution and that, if the solution consists of multiple primitives, each primitive is unambiguously identifiable.
- When the subproblem class returns multiple output primitives, its `core_problem` method should store LaTeXsnapshots of its output primitives for postprocessing reasons.

To clarify the task, we also provide the coder agent with five few-shot examples of (formatted problem specifications, python module code) pairs.

To ascertain that the code generated by the subproblem coder LLM meets the quality criteria defined in Appendix C.1, we implement three filters in a funnel, testing the most basic to the most semantic guarantees. If the output of the coder LLM passes a filter, it progresses to next one. Otherwise, the subproblem coder is given custom feedback from the filter to refine its implementation, and will need to go through the entire funnel once again. When a maximum number of coding retries is attained, the subproblem idea is **discarded**.

Code Compatibility Filter. This programmatic filter verifies that the output of the subproblem LLM coder is a well-formed Python module that correctly integrates within our framework. It runs multiple low-level automatic checks to ensure the coder's output uses correct syntax and existing objects from the `SymPy` library, conforms to our codebase's interfaces, and can be successfully executed inside the graph-sampling pipeline. The filter proceeds as follows:

- Parse the LLM coder's output and attempt to import it as a Python module.
- Instantiate the declared subroblem class and its corresponding configuration class.
- Get the `Inputs` signature exposed by the subproblem to determine its expected primitive types.
- Generate $N > 0$ independent sets of input primitive nodes for the subproblem, using the samplers of the graph generation pipeline.
- For each set of input primitive nodes, call `create_problem` to produce a subproblem node attached to the input nodes. Count a success only if the execution of `create_problem` reaches the return statement that creates an output node, without being intercepted by the problem's try/except safeguards. The `core_description` method also needs to return a Python string.

The filter passes if at least $k > 0$ of the N tests complete `create_problem` successfully and produce a string from `core_description`. If this fails, the filter provides Python tracebacks from the execution of the trials. The hyperparameters N and k are tunable.

Code Compatibility Filter: Failures Case Examples

1. `SymPy` **object hallucination**

- **Traceback**:
  [...]

are_collinear_result = sp.Boolean(p1.are_collinear(p2, p3))
AttributeError: module 'sympy' has no attribute 'Boolean'

- **Interpretation**: `SymPy` does not expose a `Boolean` class at the top level, so calling `sp.Boolean` raises an `AttributeError`. The LLM coder should import `Boolean` from `sympy.logic.boolalg` instead, otherwise `create_problem` cannot return an output node.

2. Non-robust subroblem to random input nodes, leading to failure

- **Traceback**:
  [...]
  *children = problem.create_problem(*problem._parents)*
  raise NotImplementedError("This problem requires two parallel and distinct lines as input to ensure a non-zero distance. The provided lines are either not parallel or are coincident")
- **Interpretation**: For this sub problem (`| DistanceParallelLinesProblem | [LinePrimitive, LinePrimitive] | ExprPrimitive | Compute the distance between two parallel and distinct lines |`), a non-trivial result arises only when the two input lines are parallel. Because each input line is defined by randomly sampled integer points, the probability of generating two parallel lines is negligibly small. Thus, the `create_problem` almost never returns a node output representing a distance, and almost always raises an exception handling the intersecting lines case.

3. Code does not follow our framework constraints

- **Traceback**:
  [...]
  in DistanceParallelLinesProblemConfig
  point_config=PointPrimitiveConfig(
  TypeError: PointPrimitiveConfig.__init__() got an unexpected keyword argument 'rational_coordinates'
- **Interpretation**: The problem code passes a non-existent argument to the primitive's configuration initializer, which triggers a `TypeError`.

Instruction Compliance Filter. This filter uses an LLM as a judge to review the generated code for alignment with the original instructions of the LLM coder and with the subproblem specifications from the LLM designer. It also checks for common failure cases. More precisely, it verifies that:

- The LaTeXstatement produced by the `core_description` method and the computations performed in the `create_problem` method coincide and align with the original subproblem description. The output of the `core_description` should also satisfy specific wording constraints to avoid redundant language.
- The task tackled in the subproblem admits a unique solution, is different enough from existing subproblems, and is (ideally) single-step from a `SymPy` perspective.
- The `create_problem` and `core_description` methods are both present in the subproblem class.
- The subproblem class never stores input primitive nodes as class attributes – only their `content` field (this would cause unwanted side effects in the graph generation pipeline).
- If multiple outputs exist, each should be uniquely identifiable, returned in its own tuple slot, and protected against graph re-ordering via stored LaTeXsnapshots.

If the compliance LLM judge deems that any of the above conditions is not satisfied after careful consideration, it returns "`FALSE`". Its reasoning trace is provided to the LLM coder to refine its output and potentially adjust certain problem specifications. Otherwise, the compliance LLM judge returns "`TRUE`".

Instruction Compliance Filter: Failures Case Examples

1. Subproblem Redundancy

- **LLM judge reasoning trace**: *The problem ‘PolynomialQuotientProblem‘ is highly similar in logic to the existing ‘EuclideanDivisionProblem‘. Both problems perform Euclidean division on two polynomials. [...]*
- **Interpretation**: The generated subproblem duplicates an existing one, so it should be discarded.

2. Strictly follow our framework’s constraints

- **LLM judge reasoning trace**: *The ‘core_description‘ method uses the phrase "the function expr_primitive.description()". Given that ‘expr_primitive.description()‘ for an ‘ExprPrimitive‘ typically returns "expression ‘latex_expr‘", the resulting phrase would be "the function expression ‘latex_expr‘. This is redundant and violates the condition "the description method must not use the name of a mathematical object immediately before the .description() method. [...]*
- **Interpretation**: Because the `description` method of input primitives already mentions the type of each mathematical object involved, writing “`function expr_primitive.description()`” inside the `fstring` returned by `core_description` produces redundant language.

Mathematical Relevance Filter. This last filter uses an LLM as a judge to analyze the relevance of the subproblem based on the pool of node inputs, `create_problem` outputs, and `core_description` outputs obtained during the N trials of the code compatibility filter. It ensures that:

- The subproblem is non-trivial (outputs are not constant across the code compatibility trials, not identical to the input, not empty or `None`, and of the expected `SymPy` type/shape).
- Each `create_problem` output matches the mathematically correct answer to its node input(s).
- The task implemented in the subproblem requires reasoning and yields a unique answer.
- `core_description` outputs are clear, unambiguous, and should compile in LaTeXwithout errors.

If the relevance LLM judge deems that any of the above conditions is not satisfied after careful consideration, it returns “`FALSE`”. Its reasoning trace is provided to the LLM coder to refine its output and potentially adjust certain problem specifications. Otherwise, the relevance LLM judge returns “`TRUE`”, and the subproblem code is finally considered **valid**.

Mathematical Relevance Filter: Failures Case Examples

1. Subproblem construction issue leading to triviality

- **LLM judge reasoning trace**: *The problem, as demonstrated by the provided test cases, is overwhelmingly trivial. A vast majority (85%) of the outputs are ‘Poly(1, var)‘, and the remaining 15% are ‘Poly(2, var)‘. These are all simple constant polynomials. This occurs because the generated input polynomials frequently have different variables, which leads ‘sympy.gcd‘ to compute the greatest common divisor of their constant parts or integer coefficients, resulting in a constant (often 1).*
- **Interpretation**: Our graph generation pipeline generates polynomial primitive nodes that use different variable names. Because `sp.gcd` produces meaningful results only when its operands share the same variables, the GCD here always collapses to a constant. The relevance LLM judge therefore rejects the subproblem code in its current state.

2. Non-robust subproblem to random inputs

- **LLM judge reasoning trace**: *The problem is generally acceptable, as it correctly computes the average value for various expressions and intervals. However, there are instances where the input expression is not well-defined over the given interval* *in the real domain, leading to outputs that involve unevaluated integrals or 'NaN' (Not a Number). [...] Since a significant proportion of the outputs (at least 3 out of 19, which is more than 15%) fall into this problematic category, the problem is not fully robust for all generated inputs.*

- **Interpretation**: The problem is not robust to certain input values, as visible with the random input primitive nodes sampled during the code compatibility trials. `create_problem` returns inappropriate outputs, including `NaN` values.

Motivation for the use of an agentic workflow. We delegate targeted tasks to specialized agents running separate conversations for two reasons:

- **Context hygiene:** only the information strictly needed for the each stage is provided (e.g., successful filter passes leave no trace in the LLM coder's conversation), preventing uncontrolled context growth and preserving the authority of the initial instructions [6].
- **Cost and latency:** short, independent LLM calls for the compliance and relevance filters are cheaper and faster than re-querying with the entire dialogue.

C.3 Experiments: Synthetic Subproblem Statistics

We explore the effect of a few key hyperparameters on the subproblems generated by our agentic workflow. We investigate the influence of (1) backbone LLM selection for the subproblem designer and coder agents, as well as (2) temperature T, reasoning effort R, and prompting technique used for the subproblem designer. We empirically observe that tuning T and R for the LLM coder does not impact as much the distribution of synthetic subproblems as tuning the settings of the LLM designer, so we focus on the latter only.[3]

Models. We compare Mistral's Devstral-Small-2507, OpenAI's gpt-oss-20b and gpt-oss-120b as our backbone LLMs for the designer and coder agents. This allows for assessing the impact of model size and provider on the distribution of generated subproblems. We set $T = 0.5$ and $R =$ low for the LLM designer, and $T = 0.2$ and $R =$medium for the LLM coder by default. For the subproblem designer only, we also test $T = 1$ and $R =$ high to see if these help improve subproblem diversity, novelty, and ease of implementation for the LLM coder downstream. We also experiment with the `SymPy` prompting method described in the **Subproblem Designer** paragraph of Appendix C.2 to determine whether it can help increase the variety of `SymPy` objects used in the generated problems. For the instruction compliance and mathematical relevance judges, we consistently use gpt-oss-120b with $T = 0.2$ and $R =$ medium across experiments.[4]

Metrics. To characterize the efficiency of each LLM hyperparameter combination (or **generation strategy**) for synthesizing diverse high-quality subproblems, we compute the following metrics:

- Percentage of valid subproblems generated
  $\hookrightarrow$ measures the ability of the LLM coder to implement the subproblem specifications imagined by the designer agent.
- Number of debugging iterations until generating a valid or invalid problem
  $\hookrightarrow$ quantifies how quickly the LLM coder can implement the subproblem specifications imagined by the designer agent.
- Percentage of invalid subproblems whose last fail happened due to the subproblem class being non importable or uninstantiable.
- Percentage of invalid subproblems whose last fail happened due to an insufficient percentage of successful trials in the code compatibility filter.
- Percentage of invalid subproblems whose last fail happened due to an instruction compliance rejection.

[3]This can be explained by the fact that the problem specification stage mainly dictates the new mathematical operations that will be implemented downstream in the workflow, thus directly impacting the diversity and relevance of the synthetic subproblems.

[4]Manual evaluation of the accuracy of the LLM judges for a mix of 10 subproblems (both valid and invalid) from the different backbone LLMs combined: compliance $= 8/10$; relevance $= 9/10$

- Percentage of invalid subproblems whose last fail happened due to a mathematical relevance rejection.
- Percentage of test passes (from the code compatibility trials) among valid problems
  $\hookrightarrow$ quantifies how frequently the generated subproblems can be attached to random input nodes.
- Percentage of primitives used among the ones manually implemented in advance
  $\hookrightarrow$ measures how much the subproblem designer and coder leveraged the primitives at their disposal to create new subproblems (note: some primitive classes exist only to facilitate the implementation of others, hence this metric is $< 100\%$ even for the set of manual subproblems).
- Number of unique SymPy objects used to implement the subproblem classes
  $\hookrightarrow$ quantifies the diversity of mathematical operations involved in the generated subproblems (note: this metric scales sub-linearly with the number of problems generated).
- Estimated rarity score of the SymPy objects used to implement the subproblem classes
  $\hookrightarrow$ measures how distinctive the mathematical operations involved in the generated subproblems are. We compute this by combining the value counts of SymPy objects used across all strategies, then computing their negative log frequency within the corpus of all SymPy objects observed, and finally applying logistic scaling to these values
- Average similarity of the embeddings of subproblem descriptions from their specifications
  $\hookrightarrow$ quantifies the semantic diversity of the mathematical tasks tackled in the new subproblems. We compute this by encoding the text descriptions of the subproblem specifications using the sentence encoder all-Mini-LM-L6-v2, then computing the average pairwise cosine similarity between subproblem description embeddings.

Run settings. For each generation strategy, we generate 100 new subproblems with a maximum number of debugging iterations of 10 per subproblem. The code compatibility filter executes $N = 20$ trials, with a minimum number of trial successes of $k = 3$ to pass. Metric values are obtained by averaging results over 3 random seeds.

C.4 Synthetic Subproblem Code Examples

We list below three examples of synthetic subproblem classes generated using gpt-oss-120b ($T = 0.5$, $R =$ high, SymPy prompting).

```
class MatrixTraceProblem(
    Problem[MatrixTraceProblemConfig],
    Inputs[MatrixPrimitive],
    Outputs[ExprPrimitive],
):

    def create_problem(self, matrix_primitive: MatrixPrimitive) -> Tuple[ExprPrimitive]:
        try:
            trace_expr = matrix_primitive.content.trace()
        except Exception as exc:
            raise NotImplementedError(
                f"Failed to compute trace for matrix {matrix_primitive.content!r}: {exc}"
            ) from exc
        if not isinstance(trace_expr, sp.Expr):
            trace_expr = sp.sympify(trace_expr)
        return (ExprPrimitive(content=trace_expr),)

    def core_description(self, matrix_primitive: MatrixPrimitive) -> str:
        return (
            f"Compute the trace (sum of diagonal entries) of the {matrix_primitive.description()}."
        )
```

```
class PrimeFactorsProblem(
    Problem[PrimeFactorsProblemConfig],
    Inputs[ConstantIntPrimitive],
    Outputs[VectorPrimitive],
):
    def create_problem(
        self, int_primitive: ConstantIntPrimitive
    ) -> Tuple[VectorPrimitive]:
        n = int_primitive.content

        try:
            n_int = int(n)
        except Exception as e:
            raise NotImplementedError(f"Cannot convert input to int: {n!r}") from e
        if n_int <= 1:
            factors = []
        else:
            try:
                factors = sp.ntheory.primefactors(n_int)
            except Exception as e:
                raise NotImplementedError(f"Failed to compute prime factors of {n_int}: {e}") from e
        try:
            vector = sp.Matrix(factors)
        except Exception as e:
            raise NotImplementedError(f"Failed to create VectorPrimitive from factors {factors}: {e}") from e
        return (VectorPrimitive(content=vector),)

    def core_description(self, int_primitive: ConstantIntPrimitive) -> str:
        return f"Compute the distinct prime factors of the {int_primitive.description()}."
```

```
class KurtosisProblem(
    Problem[KurtosisProblemConfig],
    Inputs[BinomialRandomVariablePrimitive],
    Outputs[ExprPrimitive],
):
    def create_problem(
        self, rv_primitive: BinomialRandomVariablePrimitive
    ) -> Tuple[ExprPrimitive]:
        rv = rv_primitive.content
        try:
            kurt = sp.stats.kurtosis(rv)
        except Exception as e:
            raise NotImplementedError(
                f"Failed to compute kurtosis for random variable {rv}: {e}"
            )
        if not isinstance(kurt, sp.Expr):
            raise NotImplementedError(
                f"Kurtosis result is not a SymPy expression: {kurt}"
            )
        return (ExprPrimitive(content=kurt),)

    def core_description(
        self, rv_primitive: BinomialRandomVariablePrimitive
    ) -> str:
        return (
            f"Compute the kurtosis of the {rv_primitive.description()}."
        )
```

C.5 Future Work

As next steps, we plan on delving into automated primitive code generation to complement new subproblems, as well as subproblem ideation from public mathematical textbooks.

D Details on the pipeline

D.1 Orchestration of the generation

To efficiently orchestrate the generation, we rely on an asynchronous pipeline where each task is handled by multiple workers (Figure 7). At its core, it uses the `vLLM` python library for fast LLM inference. This design leads to maximum GPU utilization and minimum delay before the first sample is generated.

D.2 Graph problem sampling

D.2.1 Modelization

Our approach formalizes the generation process as a graph composed of two types of nodes:

- **Primitives**, which represent symbolic mathematical objects. These are are the basic building blocks that represent inputs/outputs of problems. We consider them to be simple enough that they can be sampled directly (using heuristic methods).
- **Subproblems**, which encapsulate a well-defined mathematical operation (e.g., integration, solving a system, finding a limit...). Each subproblem takes at least one primitive as input, applies a mathematical operation to its inputs, and outputs at least one primitive. Types of the input and output primitives are part of a subproblem definition. When a subproblem is applied to existing primitives, the resulting outputs can serve as inputs to the following

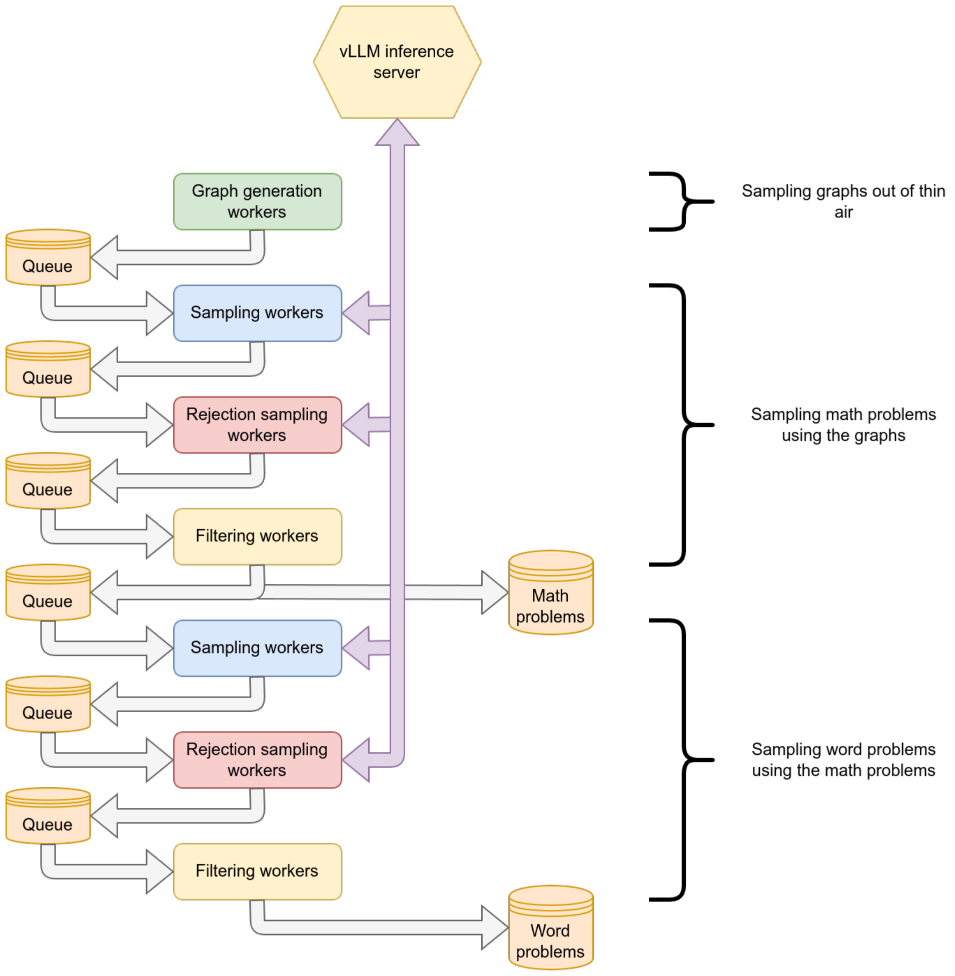

vLLM inference server
Graph generation workers
Sampling graphs out of thin air
Queue
Sampling workers
Queue
Rejection sampling workers
Sampling math problems using the graphs
Queue
Filtering workers
Queue
Math problems
Sampling workers
Queue
Rejection sampling workers
Sampling word problems using the math problems
Queue
Filtering workers
Word problems

Figure 7: End-to-end generation pipeline with asynchronous orchestration.

subproblems. This naturally leads to the graph-based sampling structure where we iteratively expand the graph with valid operations.

Type looseness, fault-tolerance. While primitives can be seen as a form of typing on mathematical objects, we do not enforce strict constraints in order to keep objects simple. Thus the pipeline is built to tolerate some operations failing. For example, a subproblem might require a matrix to be invertible, but we might not define a specific primitive for invertible matrices, thus the subproblem sampling might fail and in this case the subproblem is simply discarded from the graph.

For each subproblem, and each primitive that acts as its input, there is a directed edge from the primitive to the subproblem. Similarly, there is a directed edge from a subproblem to each of its output primitives. Thus, there are no direct edges between two subproblems. However, we allow for a direct edge between two primitives, which can either represent the construction of a recursive primitive or the construction of a coerced primitive. Also note that relative to a subproblem node, the order of input and output primitives matter, which is why our graph is not just a directed acyclic graph. A complete list of implemented subproblems and primitives, along with brief descriptions is provided in Appendices F.2 and F.1. Figure 8 shows simple example of a symbolic graph representing a problem consisting of a multiple subproblems.

Coercions. In practice, some primitives might be semantically equivalent, for instance a point in geometry will be equivalent to a vector in algebra. In these cases, they could be used interchangeably, but subproblems interfaces will enforce the use of one or the other. We allow for implementing a *coercion* mechanism that transforms one type of primitive in another type, with the goal of augmenting connectivity between different domains (Figure 2 A-5). Once implemented, these mechanisms are

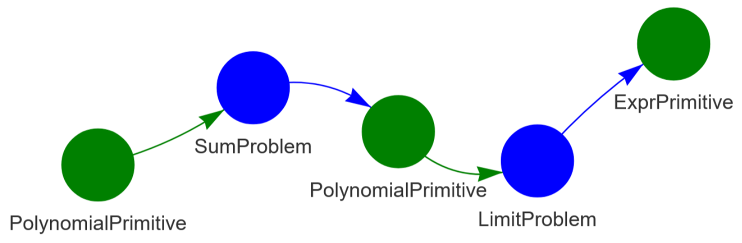


Figure 8: Green nodes represent primitives, and blue nodes represent subproblems.

used automatically in the pipeline (Algorithm 1). A list of implemented coercions is provided in Appendix F.3.

Algorithm 1 Automatic coercion of a newly added primitive. The global registry keeps track of all primitives in the current graph, sorted by type.

```
Require: New primitive $p$ of type $T$, global registry $\mathcal{P}$
1: Add $p$ to global registry $\mathcal{P}$
2: for all coercion functions $C : T \to T'$ do
3:     $p_{\text{new}} \leftarrow C(p)$
4:     if $p_{\text{new}}$ is valid then
5:         Add $p_{\text{new}}$ to registry $\mathcal{P}$
6:         Add $p_{\text{new}}$ as a child of $p$ in the graph
7:     end if
8: end for
```

D.2.2 Generation Process

The high-level generation process proceeds as follows: at each step, a new subproblem is chosen and an attempt is made to incorporate it into the graph *by either selecting compatible inputs from the existing primitives or generating new ones at random*. If the subproblem can be successfully executed with the sampled inputs, its outputs are added to the graph; otherwise, the step is discarded and the process continues. See Algorithm 2. The success of this pipeline thus depends on the ability of the subproblem to be solved with the sampled primitives. If primitives are too complex, they might not be easily used by some subproblems. Each primitive type therefore exposes some hyper-parameters that affect its sampling complexity. This sampling process results in a highly flexible generative structure. The modularity of the primitives and the openness of the sampling logic allow us to achieve a wide range of mathematical compositions.

D.2.3 Postprocessing

Once the generation process ends, we apply a few postprocessing steps to get usable graphs.

1. The graph sampling algorithm we use can result in multiple connex parts. We thus separate these different parts in as many independant problems.
2. For convenience reasons, we decide that a graph problem should only have a single leaf primitive, i.e. the whole problem leads to exactly one final answer. This requirement makes using the data for RL training easier, but could be relaxed in the future. This is achieved by a graph traversal algorithm starting from each leaf primitive.
3. We remove unused primitives that can be left over from the coercions system.
4. We apply filters to discard some problems that have unwanted features:
    - Floating numbers with too much precision
    - Symbolic objects constructed from too many operations
    - Integers that are too large

Algorithm 2 Graph generation algorithm

```
1: Initialize empty graph $\mathcal{G}$
2: Initialize empty primitive registry $\mathcal{P}$ for primitives
3: while graph size $<$ budget do
4:     Sample a new subproblem $\mathcal{T}$ from the subproblem pool (with a sampling distribution)
5:     Identify required input types for $\mathcal{T}$
6:     if primitive registry $\mathcal{P}$ is empty then
7:         generate new primitives
8:     else
9:         With probability $p$ select inputs from $\mathcal{P}$, w.p. $1-p$ generate new primitives for inputs
           (possibly recursively)
10:    end if
11:    if $\mathcal{T}$ succeeds with the selected inputs then
12:        Add $\mathcal{T}$ and its outputs to graph $\mathcal{G}$
13:        Add outputs to primitive registry $\mathcal{P}$
14:        If any inputs are not already in the primitive registry $\mathcal{P}$, add them to $\mathcal{P}$
15:    else
16:        Discard the current attempt and continue
17:    end if
18: end while
```

- *NAN* elements
- Not enough or too many subproblems

Once these postprocessing steps are applied, the generation returns the remaining graph problems.

D.2.4 Linearization

The *linearization* task, i.e. turning an abstract graph object into an equivalent textual representation, is central for the success of the method. In our initial experiments, we observed that LLMs struggled to understand the graph when its linearization was not clear enough.

Subproblems ordering The order in which the subproblems are presented plays an important role: ensuring that linearly dependent subproblems are closer together facilitates logical coherence of the downstream textual representation. With this objective in mind, we used a custom ordering method that traverses the graph by: (i) starting from a uniform random root node (i.e. a primitive with no parents); (ii) descending along a single branch until an intersection is reached; and (iii) backtracking to the root of the other branch and descending along it; all the while maintaining a record of visited nodes to avoid multiple visits to a single node.

Anonymization One value of a multi-step problem is that multiple reasoning steps are necessary to reach the final result, enabling controllable complexity. However this is true only if intermediary results, i.e. solutions to intermediary steps, are kept secret in the final problem. In practice this can be difficult because of the computational nature of the graph: subproblems tend to use their inputs in their description (e.g. "integrate f" where "f" is the solution to another subproblem).

In order to avoid subproblem solutions leaking into the textual math problem, we implement an *anonymization method* for the primitives that are the output of a subproblem. More precisely, instead of including direct symbolic outputs like : "Let $f(x) = x^3$. First, compute the derivative of $f(x)$. Then, find the minimum of $3x^2$", we want to have: "Let $f(x) = x^3$. First, compute the derivative of $f(x)$ *and call it* $g(x)$. Second, find the minimum of $g(x)$".

In terms of implementation, we use a global context that enables or disables this anonymization, so that we can generate the graph linearization (anonymization enabled) and also the final solutions (anonymization disabled). The content of a primitive is made to be dynamic depending on this global context: it can either return the true value, or an anonymized value. The anonymization also has to take into account that primitives can be recursive, in which case only part of a primitive can be anonymized (e.g. point $(x, 3)$ where x is the solution to a subproblem but not 3). This leads to a complex implementation because (i) `SymPy` is limited in what it allows with abstract variables (certain

Variables
These are the types of the different variables that appear in the computations graph:

- $b_3 \in \mathbb{R}$

Constants
These are the different constants that appear in the computations graph:

- None

Input and output nodes

- Node A: matrix $\begin{bmatrix} 2 & b_3 \\ -3 & 1 \end{bmatrix}$
- Node B: vector $\begin{bmatrix} -8 \\ 2 \end{bmatrix}$
- Node C: point $(-7,\ 8)$
- Node D: vector ν_2
- Node E: point f_2
- Node F: expression θ_3

Coercions
These are the coercions that were applied to the inputs and outputs:

- Node E was coerced from Node D: convert a vector into a point

Questions
These are all the questions numbered 1 to 2:

Question 1:

- Solve the linear system $\begin{bmatrix} 2 & b_3 \\ -3 & 1 \end{bmatrix} X = \begin{bmatrix} -8 \\ 2 \end{bmatrix}$. Input nodes: A,B. Output nodes: D.

Question 2:

- Compute the distance between the point $(-7, 8)$ and the point f_2. Input nodes: E,C. Output nodes: F.

Figure 9: Multi block description of a graph problem

objects cannot be initialized without concrete values) and (ii) some SymPy objects use the python *slot* pattern which mean that you can't easily add attributes to these objects.

Organization of information The format in which variables, primitives, relevant coercions and subproblems are specified greatly affects the LLM's ability to understand it correctly. We first experimented with a *multi block representation* (Figure 9) where primitives and subproblems were described in separate sections. Although exact, this format forced the LLM to go back and forth between blocks to recover dependencies, an operation that became fragile as the graph grew beyond three or four nodes. We therefore replaced this representation by a linear listing (Figure 10) that presents every datum exactly when it is needed. This format is lighter and thus clearer for the LLM. It retains *local context only*, as every subproblem appears with all the data it consumes or produces so the model never needs to "look up" information in another block. Also, intermediate results are reused immediately, eliminating the need for auxiliary labels. This facilitates the narrative flow as the order mirrors how a human would solve the problem.

D.2.5 Dynamic sampling

The flexibility of our approach, allowing subproblems to fail at runtime, comes at a price: the rejection probability is unknown and heterogeneous across subproblems (e.g., while it is always possible to multiply two 2×2 matrices, we expect that evaluating the limit of a randomly generated function at some point frequently fails). However it is desirable to have a method to control the distribution

Variables
These are the types of the different variables that appear in the computations graph:

- $b_3 \in \mathbb{R}$

Constants
These are the different constants that appear in the computations graph:

- None

Questions
These are all the questions numbered 1 to 2:

Question 1:

- Solve the linear system $\begin{bmatrix} 2 & b_3 \\ -3 & 1 \end{bmatrix} X = \begin{bmatrix} -8 \\ 2 \end{bmatrix}$. Input nodes: $\begin{bmatrix} 2 & b_3 \\ -3 & 1 \end{bmatrix}$, $\begin{bmatrix} -8 \\ 2 \end{bmatrix}$. Output nodes: vector ν_2.

Question 2:

- Coercion: convert the vector ν_2 into the point f_2.
- Compute the distance between the point $(-7, 8)$ and the point f_2. Input nodes: point $(-7, 8)$ and point f_2 Output nodes: expression θ_3.

Figure 10: Linearized description of a graph problem

of subproblems in the final dataset. For that purpose, we introduce a *dynamic sampling scheme that adapts on-the-fly* so that the empirical distribution converges to a target distribution (generally uniform). Our algorithm starts from a target sampling distribution, generates problems sequentially, and updates the sampling weights after each successful problem generation. The update increases the probability of subproblems with high failure rates and decreases it for subproblems that rarely fail. In order to speed up convergence, we need to account for the unique characteristics of our pipeline. Firstly, as each parallel process independently generates a set of graphs before updating the shared sampling distribution, there is a temporal mismatch between problem generation and distribution updates. Second, the entire probability mass can collapse into a single subproblem with a high rejection mass (e.g., subproblems that are very difficult to sample). In this section, we carefully examine these considerations and how we address them.

Algorithm 3 presents our dynamic sampling scheme that drives empirical distribution towards the target distribution.

Algorithm 3 Dynamic sampling with rejection feedback (retry until acceptance)

```
Require: Target distribution $(w_k^{\text{target}})_{k=0}^{K-1}$, where $K$ is number of subproblem categories
 1: Initialize sampling weights: $w_{s,k}^0 \leftarrow w_k^{\text{target}}$ for $k = 0, \dots, K-1$
 2: for $t = 0, 1, 2, \dots$ do                          ▷ advances only on accepted instance
 3:     accepted ← FALSE
 4:     while not accepted do                            ▷ retry loop
 5:         Draw category $k \sim \mathcal{P}(X_t = k) = w_{s,k}^t$
 6:         Generate concrete subproblem instance of type $k$
 7:         if subproblem is ACCEPTED then
 8:             Store subproblem in graph
 9:             accepted ← TRUE
10:         end if
11:     end while
12:     $w_{s,k}^{t+1} \leftarrow F(w_k^t, w_k^{\text{target}}, t, \lambda, \alpha)$
13: end for
```

The probability to select the subproblem k at time t is given by

$$w_{s,k}^t = \mathbb{P}(X_t = k) = \frac{1}{Z_t}\, w_k^{\text{target}}\, f\Big(\frac{w_k^{\text{target}} - w_k^t}{\lambda_t}\Big),$$

where

- t is the current time (i.e., the total number of subproblems generated up to time t).
- w_k^{target} is the target proportion for subproblem k (for now it is uniform, thus $w_k^{\text{target}} = \frac{1}{K}$).
- w_k^t is the actual proportion of subproblem k at time t.
- $\lambda_t = \frac{\lambda}{t^\alpha}$
  - $\lambda = 30$ is the regularization coefficient.
  - $\alpha = 1$ is the time penalization coefficient
- We choose the function f to satisfy the following properties:
  - $f(0) = 1$, so that $\mathbb{P}(X_t = k) = w_k^{\text{target}}$ when $w_k^{\text{target}} = w_k^t$ (perfect setup).
  - f is strictly increasing: if $w_k^{\text{target}} > w_k^t$, meaning the sample is underrepresented, its probability will increase at step $t + 1$.
  - $f(x) > 0$ for all $x \in \mathbb{R}$, since we are dealing with probabilities.

  Our choice is $f(x) = \exp(x)$.
- $Z_t = \sum_{j=0}^{K-1} w_j^{\text{target}}\, f\Big(\frac{w_k^{\text{target}} - w_k^t}{\lambda_t}\Big)$, the normalisation factor.

To address potential issue of sampling probability mass collapsing to a single subproblem, we worked on a method to improve robustness at the cost of some balance that employs clipping the sampling probabilities. This involves capping the maximum probability assigned to any subproblem type and redistributing the excess proportionally among the others, thus preventing any single subproblem from dominating the sampling process, even in the presence of high rejection weights.

Formally, we introduce a hyperparameter $c > 1$ that bounds the sampling probability of each subproblem type relative to its target weight. For every subproblem k, the probability at time t is clipped according to the following rule:

$$w_{s,k}^t \;\leftarrow\; \min\big(w_{s,k}^t,\; c \cdot w_k^{\text{target}}\big).$$

Whenever a clipping occurs for some subproblem k, we accumulate the exceeding probability mass into a residual term:

$$e \;=\; \sum_{j=0}^{K-1} \max\Big(0,\; w_{s,j}^t - c \cdot w_j^{\text{target}}\Big).$$

This excess must then be redistributed across the remaining subproblems in order to preserve normalization. We devised a method to achieve the same redistribution goal in $\mathcal{O}(K)$. For better readability, we introduce the following notation:

- $b_i = c \cdot w_i^{\text{target}}$, the clipping upper bound for subproblem i.
- $p_i = w_{s,i}^t$, the (possibly clipped) sampling probability at iteration t (time dependence omitted for clarity).
- $\Omega = \{\, i : p_i < b_i \,\}$, the active set of underrepresented subproblems eligible to receive excess mass.
- $e = \sum_{j=0}^{K-1} \max\big(0,\, p_j - b_j\big) = \sum_{j \in K \setminus \Omega} \big(p_j - b_j\big)$, the excess probability mass.
- $S = \sum_{i \in \Omega} (b_i - p_i)^2$.
- $M = \sum_{j \in \Omega} \big(b_j - p_j\big) = \sum_{j \in K} \big(b_j - p_j\big) - \sum_{j \in K \setminus \Omega} \big(b_j - p_j\big) = e + k - 1$, a normalization shortcut that arises naturally in the derivation.

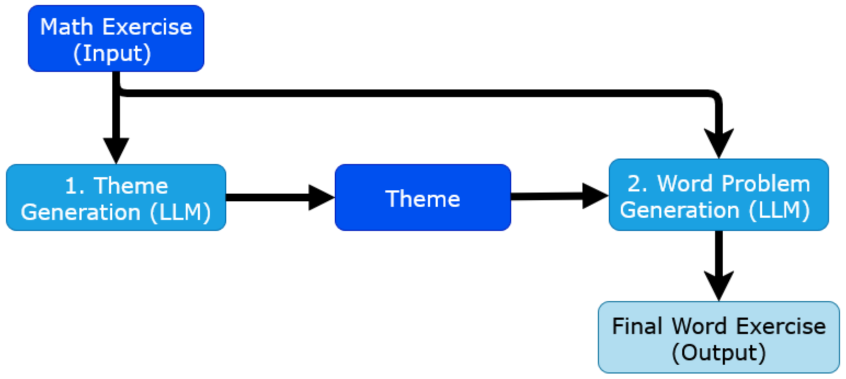

Math Exercise (Input)
1. Theme Generation (LLM)
Theme
2. Word Problem Generation (LLM)
Final Word Exercise (Output)

Figure 11: Two-step pipeline for word-problem creation

Our objective is to redistribute the excess e across Ω such that the resulting probabilities remain within $[0,\, b_i]$, while favoring underrepresented subproblems. To achieve this, we define weights $\beta_i \in [0, 1]$ satisfying the following rule:

$$p_i \leftarrow p_i + \beta_i \left(b_i - p_i \right),$$

leading to the following constraint that ensures the redistributed mass is exactly equal to the excess:

$$\sum_{i \in \Omega} \beta_i \left(b_i - p_i \right) = e. \tag{1}$$

The closed form solution for the redistribution weights is:

$$\beta_i = \frac{e + \alpha}{M} \left(1 - \lambda(\alpha) \cdot \frac{b_i - p_i}{M} \right), \quad i \in \Omega, \quad \text{where} \quad \lambda(\alpha) = \frac{\alpha \cdot M^2}{(e + \alpha)\, S}.$$

- Simple calculations show that this formula satisfies (1).
- Subproblems with larger slack $(b_i - p_i)$, that is, whose sampling probability lies below the threshold and are therefore drawn more frequently than other subproblems, receive a smaller redistribution coefficient, which is precisely the desired effect.

To ensure validity, we must enforce $\beta_i \in [0, 1]$ for every $i \in \Omega$. This leads to the following constraints on the hyperparameter α:

$$\beta_i \geq 0 \implies \alpha \leq \frac{e}{\frac{M}{S} \max_{i \in \Omega}(b_i - p_i) - 1} =: \text{A},$$

$$\beta_i \leq 1 \implies \alpha \leq \frac{(k-1)}{1 - \frac{M}{S} \min_{i \in \Omega}(b_i - p_i)} =: \text{B}.$$

Thus $0 \leq \alpha \leq \min\big(A, B\big)$. This closed-form solution avoids iterative updates entirely: once the clipped set Ω is identified, the redistribution can be computed in a single scan of size $\mathcal{O}(K)$. It also provides explicit control through the parameter α, balancing between uniform redistribution as $\alpha \to 0$ and proportional compensation as $\alpha \to \min\big(A, B\big)$, while strictly enforcing the clipping constraints.

D.3 Textual problems generation

Textual Math Problem Generation via LLM. We rely on few shot prompting to instruct an LLM to transform a linearized input graph into a coherent math problem, phrased in more natural language.

Once the textual math problem is generated, it goes through two validation steps:

- **Rejection step:** an LLM is queried to verify that the generated problem represents a coherent math problem.
- **Filtering step:** the problem is compiled to check it represents valid LaTeX.

Direct prompting A team of aerospace engineers is analyzing orbital mechanics for a new satellite deployment. They use coordinate geometry and matrix algebra to model trajectory corrections and system stability.

1. The satellite's current position is at $(-5, 8)$ in a coordinate system, while its planned trajectory follows the line connecting two reference points at $(-3, -5)$ and $(-10, 5)$. Determine the perpendicular distance β_1 from the satellite's position to this trajectory line. Express β_1 in exact form.
2. Using β_1 as the entry in the second row, first column, construct the matrix
$$\gamma_1 = \begin{bmatrix} -3 & 3 \\ \beta_1 & 5 \end{bmatrix}.$$
Calculate the matrix product $\kappa_1 = \gamma_1 \cdot \begin{bmatrix} -3 & -2 \\ 4 & -3 \end{bmatrix}$.
3. Compute the determinant x_4 of the matrix κ_1 obtained in part (2). Simplify your answer fully.

Step Back Prompting
In an ancient observatory, a celestial marker etched into the dome—corresponding to a star that aligns perfectly with the largest telescope during solstices—serves as a critical reference point located at $(-5, 8)$. To calibrate the telescope, astronomers must determine its distance to the faint, eroded path known as the **Silent Oasis Trail**, which once guided caravans between two dry wells at $(-3, -5)$ and $(-10, 5)$.

(1) Calculate the exact distance β_1 from the Astronomical Compass Point $(-5, 8)$ to the Silent Oasis Trail. This measurement will be crucial for aligning the observatory's instruments.

(2) A brass plate discovered in a sunken clockmaker's workshop bears an enigmatic matrix:
$$\gamma_1 = \begin{bmatrix} -3 & 3 \\ \beta_1 & 5 \end{bmatrix},$$
where β_1 is the result from Problem 1. This matrix represents a transformation core of a lost automaton. Multiply γ_1 by the faded maritime chart matrix:
$$\begin{bmatrix} -3 & -2 \\ 4 & -3 \end{bmatrix},$$
which sailors used to recalibrate compasses during magnetic anomalies. Denote the resulting matrix as κ_1.

(3) The automaton's mechanism depends on the determinant of the matrix κ_1 obtained in Problem 2. Compute this determinant x_4 and simplify it. This value will reveal whether the automaton's gears can operate efficiently under the calculated transformation.

Figure 12: Step-back prompting helps generate better word problems compared to direct prompting.

Word Problem Generation. Based on the generated textual math problems, we rely on prompting in order generate a corresponding word problem. The challenge is to obtain a narrative that connects potentially many distinct math subproblems. We achieve this via a step-back prompting strategy, where we first instruct the model to find coherent themes for the different mathematical objects before generating the final narrative (Figure 11). We empirically observed that this strategy leads to better results than direct prompting (Figure 12).

Similarly to their math problem counterparts, the generated word problems pass through a rejection step (where the LLM is queried to verify that the word problem is well formed) and filtering step that checks for LaTeX validity. Only problems that pass both checks are retained.

E Examples of generated problems

Math Formulation

1. A sequence of 4 independent Bernoulli trials is performed with success probability $p = \frac{1}{2}$. Compute the probability of obtaining exactly 3 successes; denote it by ε_2.
2. Determine the centre and radius of the unique circle through
$$A = (-5, 1), \quad B = (9, 7),$$
$$, \quad C = (10, 4).$$
Denote the circle by s_1, its centre by (h, k), and its radius by r.
3. Regard s_1 as a planar circle and compute its perimeter. Write
$$\sigma_1 = 2\pi r.$$
4. Convert σ_1 to an integer via
$$\tilde{\sigma} = \lfloor |\sigma_1| \rfloor.$$
Using $\tilde{\sigma}$, determine the number of ways to choose 4 distinct items from a set of $\tilde{\sigma}$ items. Denote this binomial coefficient by ρ_1.
5. Convert ε_2 to an integer similarly:
$$\tilde{\varepsilon} = \lfloor |\varepsilon_2| \rfloor.$$
Form
$$M_1 = \begin{pmatrix} 7 & \tilde{\varepsilon} \\ 7 & 1 \end{pmatrix},$$
$$M_2 = \begin{pmatrix} -5 & \rho_1 \\ \rho_1 & 4 \end{pmatrix},$$
and compute their product $z_4 = M_1 M_2$. Write z_4 explicitly.

Word Formulation

The town is preparing a grand summer festival; each task relies on the previous ones.

1. Chef Maya bakes four soufflés, each rising perfectly with probability $1/2$, independently. Determine the chance that exactly three rise perfectly (the *triple-success chance*).
2. Three historic lampposts stand at $(-5, 1)$, $(9, 7)$, and $(10, 4)$. A circular promenade must pass through all three. Find the fountain location (circle centre) and the radius.
3. Using that radius, compute the promenade's total length (perimeter), expressed as 2π times the radius.
4. Benches are placed at each whole metre around the promenade. Round the total length down to the nearest metre; this integer is the bench count. From these benches, how many different groups of four distinct benches can be chosen? (This is the *selection count*.)
5. Round the triple-success chance down to the nearest whole number (the *truncated chance*). Using these two integers, form
$$\textbf{Kitchen inventory: } \begin{pmatrix} 7 & \text{truncated chance} \\ 7 & 1 \end{pmatrix},$$
$$\textbf{Garden allocation: } \begin{pmatrix} -5 & \text{selection count} \\ \text{selection count} & 4 \end{pmatrix}$$
Multiply the two 2×2 tables and write the resulting table explicitly.

Figure 13: Parallel formulations of the same multi-step task chain: left in formal mathematics, right as an aligned real-world narrative for the festival planning scenario.

Math Formulation

Let

$$A = \{-10,\ -8,\ -4,\ -3,\ 0\}$$
$$B = \{-8,\ -5,\ -4,\ 7,\ 9\},$$

and denote by g_2 a (complex) variable. Throughout the problem the symbols e and π refer respectively to Euler's number and to the usual constant π.

1. **Cartesian product.** Determine the Cartesian product $A \times B$. Denote this set by b_1 and state its cardinality $|b_1|$.
2. **Differential equation.** Solve the linear ODE
$$-2\,\xi_2(g_2) \ - \ 4\,\frac{d}{dg_2}\xi_2(g_2) \ = \ 4,$$
subject to $\xi_2(3) = 2$. Write the solution as a function $r_2(g_2)$.
3. **Optimization.** Find the minimum value of $r_2(g_2)$ when $g_2 \in [e, \pi]$. Denote this minimum by w_3.
4. **Counting permutations.** Define
$$w_3^* = \big\lfloor\, |w_3|\, \big\rfloor.$$
Determine the number of ways to select and arrange w_3^* distinct elements from b_1 without replacement. Express your answer as an integer h_8. (Recall $P(n, k) = \dfrac{n!}{(n-k)!}$.)

Word Formulation

In a boutique bakery's kitchen, five pastry-mold trays are stamped with the numbers -10, -8, -4, -3 and 0. Five filling-container jars are labeled -8, -5, -4, 7 and 9. A pastry–filling pairing consists of one tray together with one jar.

1. Determine how many distinct pastry–filling pairings can be formed from the five trays and the five jars. State the total number of pairings.
2. The dough's rise index varies with the kitchen timer. It is described by: "Negative two times the rise index minus four times the instantaneous rate of change of the rise index per unit change in the timer setting equals four." Furthermore, when the timer reads three, the rise index reads two. Find an explicit expression giving the rise index in terms of the timer setting.
3. Consider only timer settings between "Euler hour" (e) and "Pi hour" (π). Using the expression from (2), find the smallest value the rise index attains on this interval.
4. Let the integer obtained by taking the greatest integer less than or equal to the non-negative value of the minimum rise index be the *floor integer*. Using the total number of pairings from (1), determine how many ways the baker can select and arrange that many distinct pairings on the display shelf, without repetition. Give the answer as an integer.

Figure 14: Two parallel formulations of the same tasks: left as formal mathematics, right as an equivalent real-world narrative with the same underlying structure.

Math Formulation

Variables. Let a_{13} be a (complex) variable; in part (b) we restrict a_{13} to real values. The constant e denotes Euler's number, the base of the natural exponential function.

(a) Solve the linear ordinary differential equation

$$-10\,c_{13}(a_{13}) + 7\,\frac{d}{da_{13}}c_{13}(a_{13}) \;=\; -4,$$

subject to the initial condition $c_{13}(4) = 3$. Write the solution as the function $\chi_{13}(a_{13})$.

(b) Determine the minimum value of the expression $\chi_{13}(a_{13})$ for the real variable a_{13} in the closed interval $[\,e,\;9\,]$. Denote this minimum by σ_{16} and specify the point of the interval at which it is attained.

Word Formulation

At the Central Processing Plant, Valve 13 is operated by a dual-mode control knob. The real part of the knob's setting is the visible mechanical angle, while the imaginary part records an internal electronic offset. Downstream of Valve 13 a pressure gauge measures the hydraulic pressure in bar, which depends on the knob setting.

The system obeys a linear relationship: seven times the instantaneous rate at which the downstream pressure changes per unit change of the knob setting, minus ten times the pressure itself, equals a constant net pressure drop of four bar. In other words, the pressure loss due to the friction factor (ten times the pressure) and the pressure gain due to the responsiveness factor (seven times the rate) combine to produce a net loss of four bar.

When the knob reads four, the pressure gauge reads three bar (the calibration datum).

(a) Determine the calibrated pressure curve that gives the downstream pressure for any setting of the knob, consistent with the described relationship and the calibration datum.

(b) Now consider only physically realizable knob positions, i.e., real rotations of the knob. The admissible range of knob settings runs from the reference setting equal to Euler's number (approximately 2.718 bar) up to the maximal setting nine, inclusive. Using the pressure curve obtained in part (a), find the smallest pressure value that occurs within this range and state the knob setting at which this minimum is attained.

Figure 15: Two parallel formulations of the same tasks: left as formal mathematics, right as an equivalent real-world narrative with the same underlying structure.

Math Formulation

1. Solve for the positive real number t_5 the equation

$$\frac{\ln\bigl(\ln(t_5)\bigr)}{\ln(t_5)} = 0.$$

Denote the solution by ϕ_5.
2. From a set of 7 distinct objects, in how many ways can one arrange all 7 objects without repetition? Denote this integer by γ_1.
3. Let γ_1 be the integer obtained in part (2). Consider the linear system

$$\begin{pmatrix} -6 & -5 \\ 7 & \gamma_1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -4 \\ \sinh\bigl(\cosh(e)\bigr) \end{pmatrix},$$

where e is Euler's number. Solve for the vector $X = (x, y)^T$ and denote the solution by $f_4 = (x, y)$. (Treat the integer γ_1 as an ordinary real coefficient in the matrix.)
4. Let ϕ_5 be the solution from part (1) and set

$$n := \bigl\lfloor\, |\phi_5|\, \bigr\rfloor,$$

i.e. the integer part of the absolute value of ϕ_5. Convert the vector f_4 into the point $\psi_1 = (x, y)$ with the same coordinates. Consider the two lines

L_1 : the line through $(8, 4)$ and $(2, -5)$,
L_2 : the line through ψ_1 and $(4, n)$.

Compute the smallest angle u_{14} between L_1 and L_2. Express your answer in radians, $0 \le u_{14} \le \pi$, either as an exact expression involving arccos or as a decimal approximation.

Word Formulation

1. The marine biologist keeps track of the number of days since the sea turtle Luna was released back into the ocean. The turtle's growth model leads to the condition that the ratio of the natural logarithm of the natural logarithm of the day count to the natural logarithm of the day count equals zero:

$$\frac{\ln\bigl(\ln(\text{day count})\bigr)}{\ln(\text{day count})} = 0.$$

Find the positive day count that satisfies this condition. Record this day count for use in later steps.
2. A museum curator has seven distinct artifacts that need to be displayed. In how many different ways can the curator arrange all seven artifacts? This number will be required for the next part.
3. Using the number of possible arrangements obtained in the previous step as a coefficient, solve the following system of linear equations for the coordinates of a hidden compartment beneath a garden stone slab. The system is

$$\begin{pmatrix} -6 & -5 \\ 7 & \text{number of possible arrangements} \end{pmatrix} \begin{pmatrix} \text{eastward coordinate} \\ \text{northward coordinate} \end{pmatrix} = \begin{pmatrix} -4 \\ \sinh\bigl(\cosh(e)\bigr) \end{pmatrix},$$

where e denotes Euler's number. Determine the eastward and northward coordinates of the hidden compartment.
4. The day count found in the first step has a certain magnitude. Take the absolute value of that day count and then its integer part (the greatest integer less than or equal to it). This integer will serve as a coordinate. The point representing the hidden compartment has the eastward and northward coordinates found in the previous step. Consider two garden paths:

- Path A runs through the historic gazebo at the point $(8, 4)$ and the stone fountain at the point $(2, -5)$.
- Path B runs through the hidden-compartment point and a bench located at the point $(4, \lfloor|\text{day count}|\rfloor)$.

Compute the smallest angle between Path A and Path B, measured in radians between 0 and π. Provide an exact expression involving arccos or a decimal approximation.

Figure 16: Two parallel formulations of the same tasks: left as formal mathematics, right as an equivalent real-world narrative with the same underlying structure.

Math Formulation

Let $j_1,\, n_3 \in \mathbb{Z}$.
1. Compute the finite sum

$$\phi_1(j_1) \;=\; \sum_{w_1=1}^{j_1} 7.$$

Express $\phi_1(j_1)$ explicitly as a function of j_1.
2. Using the result of part (a), evaluate the sum

$$x_9(n_3) \;=\; \sum_{j_1=1}^{n_3} \phi_1(j_1).$$

Write $x_9(n_3)$ explicitly as a function of n_3.

Word Formulation

The bakery prepares its croissants in fixed batches. Each batch is labelled with a consecutive number starting with 1 for the first batch of the morning, and each batch yields exactly seven croissants.
1. The baker decides how many batches to schedule for a particular morning. Write a formula that gives the total number of croissants produced that morning in terms of the number of batches scheduled.
2. The bakery remains open for a certain number of consecutive mornings. On the first morning the baker schedules one batch, on the second morning two batches, and so on, increasing the number of batches by one each successive morning. Using the formula from part (a), write a formula that gives the cumulative number of croissants baked over the whole period, expressed in terms of the number of mornings the bakery stays open.

Figure 17: Two parallel formulations of the same tasks: left as formal mathematics, right as an equivalent real-world narrative with the same underlying structure.

F List of primitives, problems and coercions implemented by hand

F.1 Implemented Primitives

Name	Description	Sampling Method
Expr	Expression (function) with one or multiple variables	Builds a random expression tree combining unary/binary operations, constants, variables and floats (e.g. $g(o_1) = \frac{o_1^2}{\tan(o_1)}$)
BoolExpr	Boolean expression using unions and intersections of events	Builds a Boolean-expression tree with unary (NOT) and binary (AND / OR) nodes
Point	Point in 2-D space	Randomly samples two integers as x and y coordinates
Line	Line defined by two points	Samples two PointPrimitives and constructs the line through them
Shape	Abstract geometric shape	Randomly picks a more specific shape primitive and delegates its sampling
Circle	Circle with centre and radius	Samples a PointPrimitive for the centre and a positive integer radius
Rectangle	Rectangle defined by two opposite points	Samples two PointPrimitives and constructs the rectangle
Triangle	Triangle defined by three points	Samples three PointPrimitives and constructs the triangle
Interval	Interval between two real numbers	Randomly chooses two bounds (variables, constants or floats)
CharacterPoly	Base class for parameterised polynomial primitives	Samples a degree (2–3) and returns a vector of integer coefficients
ODE	Linear ODE (subclass of CharacterPolyPrimitive)	Uses the coefficient vector to form a weighted sum of derivatives
Polynomial	Polynomial (subclass of CharacterPolyPrimitive)	Uses the coefficient vector to define the polynomial
Sequence	Numerical sequence (subclass of CharacterPolyPrimitive)	Samples an integer n and a function f, then computes $\sum_i \text{coeff}_i\, f(n+i)$
Matrix	Integer matrix	Samples size 2 or 3 and fills the matrix with random integers
Vector	Integer vector	Samples a dimension and fills the vector with random integers
FiniteSet	Finite set of integers	Samples the required number of elements (config) and forms the set
EmptySet	Empty set	Returns the empty set
FiniteOrEmptySet	FiniteSetPrimitive or EmptySetPrimitive	Returns a finite or empty set according to config weights
Bernoulli RandomVariable	Unfair coin-toss variable	Defined by probability of heads p
Binomial RandomVariable	$\text{Bin}(n, p)$ random variable	n fixed in config; p sampled with given granularity
ConstantInt	Integer constant	Random integer in $[0, \text{max\_value}]$ from config
Discrete RandomVariable	Discrete random variable	Defined by a set of outcomes sampled based on config, and probability distribution sampled based on config

(continued on next page)

(continued from previous page)

Name	Description	Sampling Method
UniformDiscrete RandomSet	Uniform RV on a set	Uniform RV on a set sampled based on config
Die	Uniform RV on $\{1, \ldots, n\}$	n defined in config

F.2 Implemented Problems

Name	Input	Output	Description
Geometry			
AngleBetweenLine	[LinePrimitive, LinePrimitive]	ExprPrimitive	Compute the angle between two lines
Area	ShapePrimitive	ExprPrimitive	Compute the area of a shape
Centroid	ShapePrimitive	PointPrimitive	Compute the centroid of a shape
CircleEquation	[PointPrimitive, PointPrimitive, PointPrimitive]	CirclePrimitive	Circle through three non-collinear points
Circumcircle	TrianglePrimitive	CirclePrimitive	Circumcircle of a triangle
Distance	[PointPrimitive, PointPrimitive]	ExprPrimitive	Distance between two points
PointLineDistance	[PointPrimitive, LinePrimitive]	ExprPrimitive	Perpendicular distance from a point to a line
LineEquation	LinePrimitive	ExprPrimitive	Equation of a line from two points
LineIntersection	[LinePrimitive, LinePrimitive]	PointPrimitive	Intersection point of two lines
LinePerpendicular	LinePrimitive	LinePrimitive	Perpendicular bisector of the segment defining the line
Perimeter	ShapePrimitive	ExprPrimitive	Compute the perimeter of a shape
Calculus / Analysis			
Differentiate	ExprPrimitive	ExprPrimitive	Derivative w.r.t. a free variable
Integrate	[ExprPrimitive, IntervalPrimitive]	ExprPrimitive	Integral of an expression
Limit	ExprPrimitive	ExprPrimitive	Limit at a singular point
Extrema	ExprPrimitive	ExprPrimitive	Min / max of an expression
ODE	ODEPrimitive	ExprPrimitive	Symbolic solution of an ODE
Sum	PolynomialPrimitive	PolynomialPrimitive	Symbolic summation over a range
Algebra & Polynomials			
Factor	PolynomialPrimitive	PolynomialPrimitive	Factor a polynomial
Solve	ExprPrimitive	SequencePrimitive	Solve an equation for a free variable

(continued on next page)

(continued from previous page)

Name	Input	Output	Description
Inequality	ExprPrimitive	IntervalPrimitive	Solve an inequality (solution interval)
Number Theory / Arithmetic			
EuclideanDivision	[PolynomialPrimitive, PolynomialPrimitive]	PolynomialPrimitive	Euclidean division (quotient and remainder)
Linear Algebra			
MulMatrix	[MatrixPrimitive, MatrixPrimitive]	MatrixPrimitive	Multiply two matrices
System	[MatrixPrimitive, VectorPrimitive]	VectorPrimitive	Solve linear system $Ax = b$
Determinant	MatrixPrimitive	ExprPrimitive	Determinant of a matrix
Characteristic Polynomial	MatrixPrimitive	PolynomialPrimitive	Characteristic polynomial of a matrix
Sequences			
Sequence	SequencePrimitive	ExprPrimitive	n^{th} term of a numerical sequence
Booleans			
ANF	BoolExprPrimitive	BoolExprPrimitive	Algebraic Normal Form
CNF	BoolExprPrimitive	BoolExprPrimitive	Conjunctive Normal Form
DNF	BoolExprPrimitive	BoolExprPrimitive	Disjunctive Normal Form
NNF	BoolExprPrimitive	BoolExprPrimitive	Negation Normal Form
AndOverOr	BoolExprPrimitive	BoolExprPrimitive	Distribute AND over OR
OrOverAnd	BoolExprPrimitive	BoolExprPrimitive	Distribute OR over AND
Satisfiability	BoolExprPrimitive	BooleanPrimitive	Satisfiability test
Set Problems			
SetIntersection	[FiniteOrEmptySet, FiniteOrEmptySet]	FiniteOrEmptySet	Intersection of two sets
SetUnion	[FiniteOrEmptySet, FiniteOrEmptySet]	FiniteOrEmptySet	Union of two sets
CartesianProduct	[FiniteOrEmptySet, FiniteOrEmptySet]	FiniteOrEmptySet	Cartesian product of two sets
SetCardinality	FiniteOrEmptySet	ExprPrimitive	Cardinality of a finite set
Counting Problems			
SetCountPermutation	[FiniteOrEmptySet, ExprPrimitive]	ExprPrimitive	Number of k-permutations of a set
CountPermutation	[ConstantIntPrimitive, ConstantIntPrimitive]	ConstantIntPrimitive	Number of k-permutations of n elements
SetCountCombination	[FiniteOrEmptySet, ExprPrimitive]	ExprPrimitive	Number of k-combinations of a set

(continued on next page)

(continued from previous page)

Name	Input	Output	Description
CountCombination	[ConstantIntPrimitive, ConstantIntPrimitive]	ConstantIntPrimitive	$\binom{n}{k}$ combinations
Random Variable Problems			
ExpectationRV	DiscreteRVPrimitive	ExprPrimitive	Mathematical expectation of a random variable
SumRandomCoin	[BernoulliRVPrimitive, BernoulliRVPrimitive]	ExprPrimitive	Probability of an outcome when tossing two unfair coins
SequenceCoinTosses	BernoulliRVPrimitive	ExprPrimitive	Probability of a given H/T sequence in repeated tosses
Binomial Prob Num Successes	ConstantIntPrimitive	ExprPrimitive	Probability of k successes (input) for $\mathrm{Bin}(n, p)$ where p is sampled according to config
RollDieDivisible	DiePrimitive	ExprPrimitive	Probability that a roll is divisible by k on an n-sided die
Roll Die Optimal Stopping	DiePrimitive	Union[ExprPrimitive, FiniteOrEmptySet]	Returns expected payoff of the optimal strategy in the game where we can roll a die at most k times (sampled randomly based on config). Or returns set of outcomes that makes it optimal decision to stop the game after first draw. The version of problem picked randomly based on the config.
Brainteasers			
BurningRope	ConstantIntPrimitive	Union[ExprPrimitive, FiniteOrEmptySet]	Famous burning rope puzzle, number of ropes provided as input. Returns the closest time to unit we can measure with the number of ropes provided, or the set of all times strictly less than unit we can measure with these ropes. The version of problem picked randomly.

(continued on next page)

(continued from previous page)

Name	Input	Output	Description
GreedyPirates	ConstantIntPrimitive	VectorPrimitive	Famous greedy pirates puzzle, where a number of pirates (input) is splitting a number of coins (config). Returns a vector of rewards allocated to two most senior pirates once the optimal proposal is made at the stage where (randomly picked) number of players have already been eliminated from the game.
Quant Interview Style			
DerivativePrice	DiscreteRVPrimitive	ExprPrimitive	Input random variable represents future payoff of an asset k steps ahead (k in config). Determine present value of the expectation of asset derivative defined by a function from config, if the time value of money is determined by a one step ahead discounting factor (config).
DerivativePriceNew	[DiscreteRVPrimitive, ExprPrimitive]	ExprPrimitive	Input random variable represents future payoff of an asset k steps ahead (k in config). Determine present value of the expectation of asset derivative defined by input function, if the time value of money is determined by a one step ahead discounting factor (config).

F.3 Implemented coercions

Input	Output	Description
PolynomialPrimitive	ExprPrimitive	Transform the polynomial into an expression (100% success)
ExprPrimitive	PolynomialPrimitive	Convert expression into a polynomial

(continued on next page)

(continued from previous page)

Input	Output	Description
SequencePrimitive	PolynomialPrimitive	Sequence $\rightarrow$ polynomial via characteristic polynomial
PolynomialPrimitive	SequencePrimitive	Polynomial $\rightarrow$ sequence via characteristic polynomial
PolynomialPrimitive	ODEPrimitive	Polynomial $\rightarrow$ ODE via characteristic polynomial
VectorPrimitive	PointPrimitive	Size-2 vector $\rightarrow$ 2-D point
PointPrimitive	VectorPrimitive	2-D point $\rightarrow$ size-2 vector
FiniteSetPrimitive	FiniteOrEmptySet	Finite Set -> Finite or empty set
EmptySetPrimitive	FiniteOrEmptySet	Empty Set -> Finite or empty set
ConstantIntPrimitive	ExprPrimitive	Integer -> Expression
ExprPrimitive	IntPrimitive	Expression -> Integer by taking floor of its absolute value
PolynomialPrimitive	IntPrimitive	Degree of a polynomial to integer
MatrixPrimitive	IntPrimitive	Size of matrix to integer
FiniteSetPrimitive	IntPrimitive	Size of a set to integer
UniformDiscrete RVSet Primitive	DiscreteRVPrimitive	Uniform discrete random variable into a discrete random variable
BinomialRVPrimitive	DiscreteRVPrimitive	Binomial random variable into a discrete random variable
DiePrimitive	Uniform Discrete RV Set Primitive	Random variable representing roll of a die into a uniform discrete random variable
DiePrimitive	DiscreteRVPrimitive	Random variable representing roll of a die into a discrete random variable