MultiLingPoT: Boosting Mathematical Reasoning in LLMs through Multilingual Program Integration

Nianqi Li[1], **Zujie Liang**[2], **Siyu Yuan**[3], **Jiaqing Liang**[3], **Feng Wei**[2], **Yanghua Xiao**[1]*
[1]Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University
[2]MYbank, Ant Group [3]School of Data Science, Fudan University
nqli23@m.fudan.edu.cn, shawyh@fudan.edu.cn

Abstract

Program-of-Thought, which aims to use program instead of natural language in reasoning, is an important way for LLMs to solve mathematical problems. Since different programming languages excel in different areas, it is natural to use the most suitable language for solving specific problems. However, current research only focuses on single language PoT, ignoring the differences between programming languages. Therefore, this paper proposes a multilingual programme reasoning method, MultiLingPoT, and deeply explores the impact of multilingual integration in the training and inference. This method allows the model to answer questions using multiple languages by fine-tuning on multilingual data and improving individual language's reasoning accuracy by 2.5%. Additionally, prior and posterior selection methods are used to help the model select the most suitable language during inference, and achieves 8% performance gains. Finally, our code metric analysis shows that language differences manifest in encapsulation levels and implementation granularity, while strategic deviation from language conventions can enhances code performance.[1]

1 Introduction

Program-of-Thought (PoT), which aims to use programming language instead of natural language as an intermediate step in reasoning in Large Language Models (LLMs), is an important way for LLMs to solve mathematical problems (Chen et al., 2022; Gao et al., 2023). By generating programs and running them with a code interpreter, LLMs can not only exploit their reasoning capabilities but also avoid the computational errors. Inspired by mathematical tasks, nowadays, PoT is also widely used in other domains, such as pseudo-code graph (Skianis et al., 2024), visual inference (Surís et al., 2023), and document understanding (Zhao et al., 2024). Therefore, exploring the optimization of PoT is a problem of great value.

```
Find the modular inverse of 3 modulo 11.
```

```
import sympy as sp                                  Python PoT
inverse = sp.mod_inverse(3, 11)
```

```
int a = 3, m = 11; int x, y...                      C++ PoT
while (b != 0) {
    int q = a1 / b; int temp = a1 % b;
    a1 = b; b = temp; temp = x1 - q * x2;
    x1 = x2; x2 = temp; temp = y1 - q * y2;
    y1 = y2; y2 = temp;
} int result = (x1 % m + m) % m;...
```

```
Given the matrix A and the vector v, compute
the matrix product A·v.
```

```
for (int i = 0; i < rows; i++) {                    Java PoT
    int sum = 0;
    for (int j = 0; j < cols; j++) {
        sum += A[i][j] * v[j];}
    result[i] = sum;...
```

```
A = [2 1; 3 4]; v = [1; 2];                         Matlab PoT
result = A * v;
```

Figure 1: Examples of different programming languages having different advantages.

Previous work on PoT has concentrated on enhancing the mathematical capabilities based on a single language PoT, including techniques such as data augmentation (Yue et al., 2023; Jie and Lu, 2023), PoT-CoT integration, and multi-round iteration (Wang et al., 2023b; Gou et al., 2023; Qian et al., 2023). However, we believe that **different programming languages have different specializations**. As shown in Figure 1, matlab is better at matrix operations than Java, while Python provides a rich library for number theory. Therefore, it is a better approach to use the suitable language to solve the corresponding problem than to use one

*Corresponding author.
[1]Resources of this paper can be found at https://github.com/Nianqi-Li/MultiLingPoT

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 19794–19811
November 4-9, 2025

language to solve all problems. Recently, Luo et al. (2024) also started to explore combining different programming languages. However, by voting for better performance, they only take different languages as a kind of diversity enhancement, and do not deeply analyze the differences and impacts of language characteristics. Meanwhile, their research is limited to the prompt level, and lacks the exploration of multi-language training. Therefore, the use of multi programming languages in training and reasoning still deserves in-depth research.

To fill the gap and further enhance the mathematical reasoning ability of LLM, we explore three questions: 1) Q1: During training, can the fine-tuning of the multi-language improve the model's mathematical reasoning ability? 2) Q2: During inference, does strategically selecting languages for problem-solving enhance the model's mathematical reasoning performance? 3) Q3: What brings the differences between languages and what defines language-specific code effectiveness?

To answer the above questions, we introduce MultiLingPoT, a multilingual program reasoning framework, which allows the model to solve mathematical problems using PoT in multiple languages. For Q1, we construct a large amount of multilingual programming data using ChatGPT (OpenAI, 2022) and fine-tune the MultiLingPoT model based on our high-quality PoT data. Compared to single language model, we find that MultiLingPoT is able to learn from the strengths of different languages, showing a stable improvement of 2.5% on average. For Q2, we design a series of prior and posterior selection strategies, including self-consistency, in-context learning and reward model to select a suitable programming language for MultiLingPoT. The results show that the appropriate selection strategy can further improve the math ability of MultiLingPoT, with a 8% improvement compared to the best direct output. Finally, for Q3, we analyze differences across languages and their preferences in accurate solutions using software code metrics. Based on this, we provide language-specific suggestions to generate more accurate result.

The main contributions of this paper are summarized as follows:

- We propose MultiLingPoT, an multilingual PoT method that enables the model to answer mathematical questions using multiple programming languages and select the optimal language for each problem.
- We find that models benefit from multi-languages training, achieving a 2.5% improvement over single-language fine-tuning.
- We find that appropriate language selection in inference is better than answering in a single language, showing an improvement of 8% compared to the best direct output.
- We analyze differences between correct/erroneous code across languages and provide practical suggestions to improve code accuracy.

2 Related Work

Program-of-Thought Program-of-Thought, which aims to use programming language instead of natural language in LLMs' reasoning, is an important way to solve mathematical problems. By transferring the computation to code interpreter, LLMs are able to avoid computational errors while exploiting reasoning capabilities. In 2022, Chen et al. (2022) introduced the use of code as an intermediate step to assist LLMs, while Gao et al. (2023) proposed the program-aided language model. By building PoT data and fine-tuning, LLMs are able to enhance their mathematical capabilities (Yue et al., 2023; Jie and Lu, 2023; Luo et al., 2023; He-Yueya et al., 2023). Further, some research combine Chain-of-Thought's(COT) reasoning with PoT's computation as tool use (Ma et al., 2025a), resulting in models such as MathCoder (Wang et al., 2023b) and ToRA (Gou et al., 2023). However, these studies are limited to Python-based PoT, ignoring the differences between different programming languages. Although, recent Luo et al. (2024) start to explore combining different languages. They only use different languages as an enhancement of diversity, without deeply analyzing the impacts of different language characteristics. Therefore, this paper proposes MultiLingPoT, an multi-programming language reasoning approach that explores multi-language integration in training and inference, and analyzes the language differences in view of the code metrics.

Multiple Programming Languages Multi-program languages processing is a classic research topic in code intelligence. It has been studied for a long time in tasks such as code repair (Luo et al., 2025), code search (Wang et al., 2023a), code summarization (Yang et al., 2025) and comment generation (Ahmed and Devanbu, 2022).

For instance, multilingual encoders are trained for cross-lingual comment generation (Yang et al., 2025), and weak-language performance is enhanced via inter-language translation (Luo et al., 2025). However, there are fundamental differences between multilingual research on PoT and code intelligence. While code intelligence focuses on code understanding and generation (Wan et al., 2024), PoT emphasizes problem-solving through code execution, requiring task understanding, reasoning, and computational planning. In addition, multilingual code intelligence prioritizes cross-lingual generalization, whereas PoT demands language-integrated problem-solving. Therefore, although multilingual research has been explored in code intelligence, the multi-languages problem in PoT remains an unexplored domain.

3 MultiLingPoT

In this section, we present the construction of MultiLingPoT to answer three key questions, including foundation language selection, data construction, model training, model inference, and code analysis. The illustration is shown in Figure 2.

3.1 Foundation Languages

To effectively implement and evaluate MultiLingPoT, foundation languages are of crucial importance. Based on heuristic thinking, the chosen programming language should have the following characteristics: 1) The syntactic differences between languages should be significantly distinguishable so as to separate MultiLingPoT from single-language reasoning. For example, C and C++ are not a good choice. 2) The foundation languages should be popular, to avoid bias due to disparities in the model's proficiency across languages. 3) The language should support mathematical reasoning. For example, HTML, which is designed for the web, is not a suitable choice. Based on the above criteria and following GitHub's (2024) report, Python, C++, Java, and Matlab are established as the foundation languages, providing a solid basis for the experiments.

3.2 Data Construction

In order to teach the model to answer questions in multilingual programs, we construct training data across various languages. Based on related work (Jie and Lu, 2023; Yue et al., 2023; Luo et al., 2023), we select GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) as the foundation for dataset construction, and used ChatGPT (OpenAI, 2022) to generate PoT data in multiple languages. Specifically, for each problem in the trainsets, we instruct ChatGPT to generate solutions in four programming languages: Python, C++, Java, and Matlab. To improve ChatGPT's ability to generate program in each language, we provide four manually crafted solution program as examples, which are provided in Appendix A.1. Due to the varying difficulty of GSM8K and MATH, different sets of examples are used for each, enabling better generation performance. Finally, to ensure the quality of the training set, we execute all the generated programs using code interpreters. Only the programs with correct results are kept.

Dataset	Origin	Python	C++	Java	Matlab
GSM8K	7473	6598	6535	6615	6612
MATH	6282	3844	3737	3575	3619

Table 1: Constructed PoT data for Python, C++, Java and Matlab based on GSM8K and MATH.

Table 1 shows the results of our constructed dataset. We collect 26,359 samples for GSM8K and 14,775 samples for MATH. Since MATH is more challenging and has more erroneous outputs, the dataset for MATH is smaller than GSM8K. However, as the data across different languages is balanced in both datasets, our data is suitable and fair for multilingual programming training.

3.3 Model Fine-Tuning

For Q1, to examine the impact of multi-language PoT in model training, we develop the MultiLingPoT model based on our multilingual programming dataset, enabling mathematical problem-solving across various programming languages. For an input query and a specified programming language, the MultiLingPoT model outputs a function in the corresponding language called "solution", which returns the result of the query. For more details, the instruction and input templates are provided in Appendix A.2. Parameter settings and model selection for training are provided in Section 4.1.

3.4 Inference Language Selection

For Q2, to select the appropriate language during inference, we design a series of different selection strategies to verify the validity of the selection. Based on the time and input, we categorize the selection strategies into prior and posterior. Figure 2

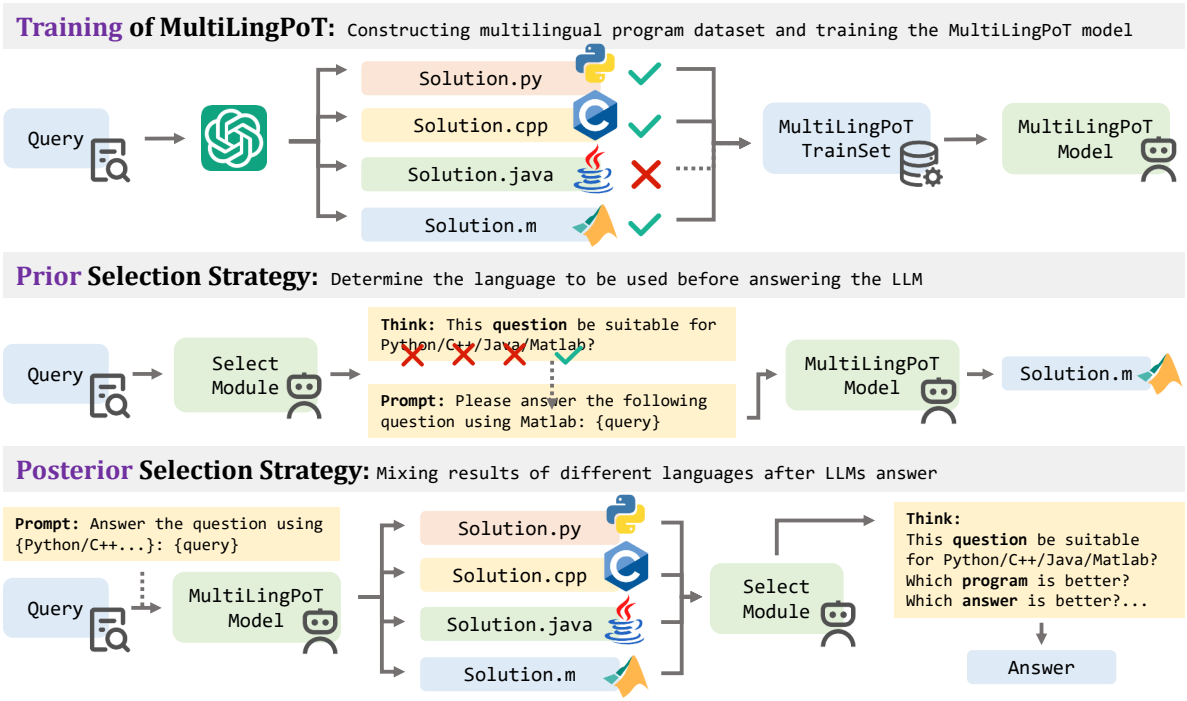


Figure 2: The illustration of the implementation of the MultiLingPoT methodology, including data construction, model training and the selection strategies. Considering the diverse implementations of selection strategies, the "Think" part only represents the underlying logic of the selection strategy, but not its specific implementation.

illustrates these two types of selection.

Prior Selection Strategy Prior selection is a selection strategy only based on the query. This strategy determines the language to be used before the LLM generates the response, requiring only one generation. Four specific implementations are explored under this strategy: 1) Case-Based Choice, selecting a language based on similar queries. 2) Small Reward Model, using a small parameter model such as Bert (Devlin, 2018) as the reward model for programming language selection. 3) LLM Reward Model, using a large model such as Llama3 (AI@Meta, 2024) as a reward model for programming language selection. 4) Direct Perference Optimization, training MultiLingPoT model to select using preference learning.

Posterior Selection Strategy Posterior selection is a selection strategy based on the query and generated code in four languages. It is used after MultiLingPoT generates the four language solutions to select the best result. Compared to the prior selection, the posterior selection has more information. However, since each query requires four rounds of inference to generate answers in different programming languages, the posterior inference requires more computation time. Four specific implementations are explored under this strategy: 1) Self-Consistency, selecting the final result by voting. 2 & 3) Reward model, using a small parameter model or LLM as the reward model for programming language selection. 4) Voting and Reward, using LLM reward model for tie-breaking options.

More implementation details including case quantities, reward scoring metrics, and preference pair construction are provided in the Appendix C.2.

3.5 Code Analysis

For Q3, we move beyond surface-level accuracy and analyze code using software metrics to explore intrinsic differences between languages. We employ 9 metrics categorized into 3 types: Size (length, rows, operations), Complexity (conditionals, loops, nesting, cognitive complexity), and Structure (encapsulation, function calls) (Riguzzi, 1996; Timóteo et al., 2008; Nuñez-Varela et al., 2017). Base on these metrics, we analyze cross-language metric variations to quantitatively reveal language-specific characteristics. Further, by analyzing the dynamics of the correct code metrics, we identify key quality features and suggest language-specific optimizations for code generation.

Method	Language	GSM8K	SVAMP	NumG	Mat	ASDiv	Average
SinglePoT	Python	64.06	71.70	44.58	54.07	72.60	61.40
	C++	64.97	71.50	43.30	33.08	71.83	56.93
	Java	63.00	72.80	43.01	42.90	75.00	59.34
	Matlab	62.62	69.70	42.30	34.55	71.83	56.20
	Python-DA	64.36	73.80	42.45	52.50	**77.54**	62.13
MultiLingPoT	Python	65.57	73.10	45.44	53.23	73.65	62.19
	C++	64.97	73.60	45.44	37.16	73.70	58.97
	Java	67.02	75.10	44.87	45.19	73.80	61.19
	Matlab	65.42	73.30	44.01	37.89	72.26	58.57
	Self-Cons.	**69.37**	**75.60**	**46.72**	**54.69**	75.09	**64.29**

Table 2: Results of MultiLingPoT training on simple datasets. "-DA" indicates data augmentation, and "Self-Cons." refers to Self-Consistency. indicates in-domain testing, and indicates out-of-domain testing

Method	Language	Algebra	Count Prob.	Geom.	Int. Algebra	Num. Theory	Prealg.	Precalc.	Average
SinglePoT	Python	35.94	24.89	17.69	15.55	48.17	45.51	21.54	29.89
	C++	37.85	27.42	21.10	16.73	39.73	46.59	19.37	29.82
	Java	39.30	31.85	19.61	19.21	43.76	48.38	23.24	32.19
	Matlab	29.48	34.59	18.12	11.50	40.69	43.72	19.61	28.24
	Python-DA	39.94	31.43	22.60	21.56	52.01	47.90	**26.39**	34.54
MultiLingPoT	Python	38.85	26.16	21.10	19.86	48.94	49.58	21.54	32.29
	C++	41.12	30.80	21.53	18.95	42.80	52.56	22.03	32.82
	Java	40.94	33.33	23.66	18.82	47.21	51.97	22.51	34.06
	Matlab	31.11	36.70	20.89	12.28	46.06	47.55	19.85	30.63
	Self-Cons.	**45.04**	**37.76**	**23.88**	**23.13**	**53.16**	**57.34**	24.21	**37.78**

Table 3: Results of MultiLingPoT training on complex datasets.

4 Experiments

In this section, we first explore MultiLingPoT's performance from both training and inference perspectives to answer Q1 and Q2. Then, we measure the code metrics across language and response correctness for Q3. In addition, we test MultiLingPoT on multiple models to examine its generalisability.

4.1 Experiments Setup

Training Setup We perform full fine-tuning of `CodeLlama-7B-hf` (Roziere et al., 2023) on the dataset constructed in Section 3.2 to obtain the MultiLingPoT model. During training, we set the learning rate to 2e-5, the global batch size to 128, and the maximum sequence length to 1024 for three epochs. To accelerate training, we use DeepSpeed ZeRO Stage 3 (Rajbhandari et al., 2020). All training operations are performed using Llama-Factory (Zheng et al., 2024).

Evaluation Setup Since the difficulty of the problem affects the training and inference of MultiLingPoT, we test it on both simple and complex datasets. For the simple dataset, we train on the GSM8K variant dataset from Section 3.2, and additionally use SVAMP (Patel et al., 2021), NumGLUE (Mishra et al., 2022), Mathematics (Davies et al., 2021), and ASDiv (Miao et al., 2021) for evaluation. For the complex dataset, we train on the MATH variant dataset from Section 3.2 and test on seven categories from the MATH testset: Algebra, Counting & Probability, Geometry, Intermediate Algebra, Number Theory, Prealgebra, and Precalculus. All the evaluations use accuracy as the metric.

4.2 *A1: MultiLingPoT Training Enhances Mathematical Reasoning*

To answer Q1, we evaluate MultiLingPoT's mathematical reasoning performance across languages after fine-tuning, and compare it against single-language PoT baselines, which are fine-tuned with single language dataset in Section 3.2. Further, we perform four times data augmentation of python as a stronger baseline. Tables 2 and 3 show the results.

Overall, we have the following conclusions: 1) Languages exhibit distinct task-specific strengths.

Method		GSM8K	SVAMP	NumG	Mat	ASDiv	Average
SinglePoT	Python-DA	64.36	73.80	42.45	52.50	77.54	62.13
	Python-DA.SC	67.24	74.00	43.58	56.47	**78.11**	63.88
MultiLingPoT Prior	Case-Based Choice	65.35	73.40	45.01	50.93	74.23	61.78
	Bert RM	66.18	74.10	44.15	42.58	73.75	60.15
	CodeBert RM	65.88	73.30	44.30	37.99	72.21	58.73
	Llama3 RM	64.82	73.70	45.44	36.95	72.93	58.76
	DPO	62.85	70.70	42.87	39.56	71.83	57.56
MultiLingPoT Posterior	Self Consistency	69.37	75.60	46.72	54.69	75.09	64.29
	Bert RM	65.65	74.20	45.01	40.81	73.22	59.77
	CodeBert RM	66.86	75.30	43.87	44.78	73.27	60.81
	Llama3 RM	**72.55**	**78.00**	**48.43**	55.53	77.59	**66.42**
	Voting & Reward	70.88	77.90	47.43	**56.78**	76.19	65.83
	Random	64.82	72.50	44.30	43.31	72.88	59.56
	Upper Bound	79.37	83.70	53.84	64.61	81.09	72.52

Table 4: Results of MultiLingPoT with selection strategies on simple datasets. "RM" indicates reward model. "Random" and "Upper Bound" are for MultiLingPoT and are provided as baselines.

Method		Algebra	Count Prob.	Geom.	Int. Algebra	Num. Theory	Prealg.	Precalc.	Average
SinglePoT	Python-DA	39.94	31.43	22.60	21.56	52.01	47.90	26.39	34.54
	Python-DA.SC	44.94	33.54	23.02	25.22	56.04	52.80	**26.63**	37.45
MultiLingPoT Prior	Case-Based	43.03	33.12	22.60	21.56	48.75	52.44	24.45	35.13
	Bert RM	38.76	35.65	22.60	20.39	51.24	51.73	21.54	34.55
	CodeBert RM	40.40	34.59	21.74	21.30	49.71	51.85	23.97	34.79
	Llama3 RM	36.94	31.22	21.10	18.56	45.48	53.64	21.54	32.64
	DPO	33.57	36.28	17.91	18.16	45.10	42.17	20.33	30.50
MultiLingPoT Posterior	Self Consistency	45.04	37.76	23.88	23.13	53.16	57.34	24.21	37.78
	Bert RM	43.22	35.86	22.17	23.39	51.24	52.92	23.00	35.97
	CodeBert RM	39.67	35.65	23.45	23.13	49.52	53.64	23.24	35.47
	Llama3 RM	41.40	41.98	24.09	25.88	55.47	59.49	24.93	39.03
	Voting & Reward	**47.95**	**42.40**	**24.52**	**26.01**	**57.00**	**59.73**	26.15	**40.53**
	Random	34.03	30.16	22.60	16.99	45.87	51.37	21.30	31.76
	Upper Bound	58.32	53.37	34.11	34.37	68.13	68.45	33.89	50.09

Table 5: Results of MultiLingPoT with selection strategies on complex datasets.

For example, C++ is good at GSM8K, while Matlab shows a clear advantage in Counting & Probability. Word cloud analysis in Figure 3 further visualizes these linguistic divergences. And the Appendix C.1 shows some examples. 2) Training with MultiLingPoT enhances reasoning performance across all languages. Compared to SinglePoT, each language in MultiLingPoT improves about 2%, which proves that different languages can learn from each other. 3) Compared to data-augmented SinglePoT, MultiLingPoT still demonstrates its superiority. Among all 12 test sets, MultiLingPoT achieves higher accuracy in 8 of them, demonstrating that multilingual training improves both cross-lingual generalization and accuracy enhancement.

Furthermore, comparing different difficulty levels, we find that multilingual training provides greater benefits on simple datasets. Since data augmentation offers limited diversity for simple tasks, while cross-language has better diversity, python in MultiLingPoT even outperforms python PoT with data enhancement in simple testset. On hard datasets, by applying self-consistency, MultiLingPoT can still achieves better performance than data augmentation.

4.3 *A2: Inference Language Selection Optimizes MultiLingPoT*

To answer Q2, we test the performance of the language selection strategy in Section 3.4. Tables 4 and 5 show the results. And the Appendix C.2 shows some cases.

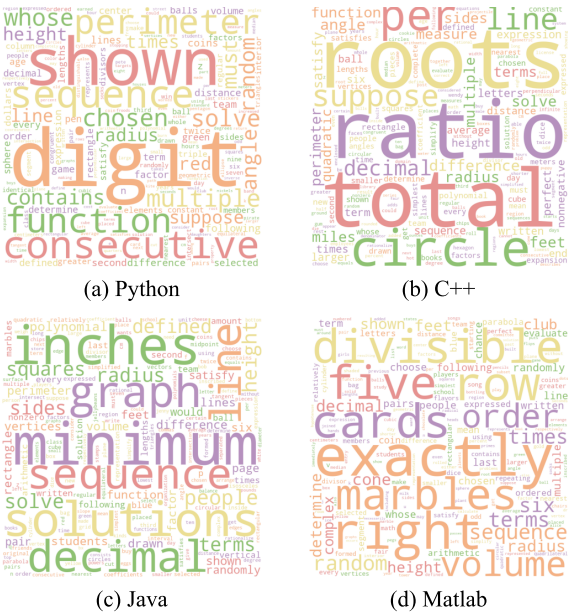
Overall, our findings are as follows: 1) A suitable language selection strategy can significantly improve model inference performance. The opti-

Language	Length	Rows	Oper.	Conds.	Loops	Nesting	Cognitive	Encap.	FCalls
Python	232.99	8.73	6.41	0.31	0.51	0.58	1.29	**1.13**	1.80
C++	**342.33**	**12.71**	**12.28**	0.39	0.52	3.67	13.25	1.11	0.88
Java	282.34	12.30	7.25	**0.40**	**0.55**	**3.75**	**14.41**	1.05	1.29
Matlab	219.96	8.48	4.61	0.23	0.37	0.65	1.41	1.02	**2.05**

Table 6: Results of software code metrics for different programming languages in problem-solving.

Language	Length	Rows	Oper.	Conds.	Loops	Nesting	Cognitive	Encap.	FCalls
Python	**-9.79**	-5.32	-3.97	**12.10**	-1.58	7.79	**16.12**	1.82	**-17.03**
C++	**-15.22**	**-9.50**	**-29.39**	-5.58	-2.11	2.43	-5.72	2.47	**8.21**
Java	-2.25	-1.73	-3.89	-3.95	1.08	0.19	0.36	0.19	-5.96
Matlab	-2.70	-0.33	**-23.60**	**28.36**	**33.77**	**23.61**	**18.85**	2.35	**-15.86**

Table 7: Characteristic preferences of correct codes in different programming languages. All results are calculated in percentages, indicating deviations of correct codes to the average. Differences greater than 8% are **bolded**.



(a) Python (b) C++

(c) Java (d) Matlab

Figure 3: Word clouds for four programming languages show key terms in their specialized problem areas.

mal selection strategy improves average accuracy by 8% compared to no selection inference. Even when evaluated against data-augmented SinglePoT with 4-path self-consistency, our strategy maintains a 3% advantage. It shows that the language selection strategy can identify areas of strength in different languages, which is more reasonable and effective than the integration of sampling in a single language. 2) Posterior selection is better than prior selection. Since posterior selection has more information, it improves performance by about 4% over prior selection on both simple and complex datasets, which shows that the design of the selection strategy can greatly affect the performance of inference. 3) Difficult problems are more suitable for language selection. For all selection strategies, the improvement in the complex dataset is 2% higher than the simple dataset. It is even more obvious that prior selection is almost ineffective in the simple dataset while it is significantly improved in the complex dataset. This may be due to the fact that complex queries have more language preferences, such as matrix operations and solving equations, which suggests that complex problems have more potential for language selection in reasoning. 4) There is still room for improvement in language selection strategies. Despite exploring multiple strategies across simple and complex datasets, the current best strategy still lags behind the theoretical upper bound.

4.4 *A3: Abstraction Levels Shape Code Paradigms and Quality*

To address Q3, we first test the performance of different languages in problem-solving using software code metrics in Section 3.5. The results are presented in Table 6. Among them, C++ and Java exhibit larger size, with longer code length, more lines, and greater operation demands. Correspondingly, these two languages demonstrate higher complexity, primarily reflected in increased nesting levels. For code structure, encapsulation levels vary minimally across languages, while Python and Matlab show more frequent function calls. In summary, this reflects two distinct coding paradigms: low-abstraction languages like C++ and Java tend to build solutions from the ground up, whereas high-abstraction languages like Python and Matlab often leverage their rich library ecosystems to achieve goals through function calls.

Furthermore, we examine characteristic prefer-

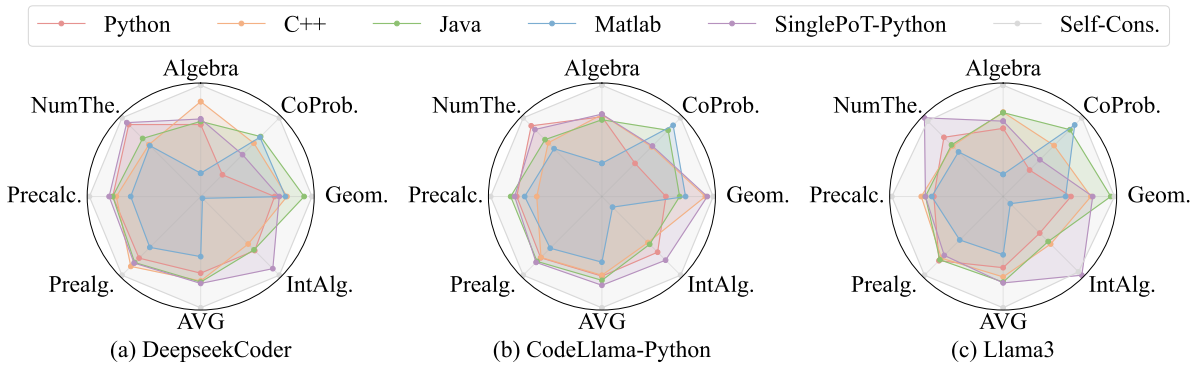


Figure 4: Results of different models using MultiLingPoT on complex datasets, including DeepseekCoder for the code model, CodeLlama-Python for the code model in a single language, and Llama3 for the non-code model.

ences of correct code. We select medium difficulty questions containing both incorrect and correct solutions for analysis. Additionally, relative values are adopted to eliminate inherent metric differences across programming languages. Specifically, we calculate the percentage deviation between correct code and the average performance. Our findings are presented in Table 7.

For size, correct code generally favors smaller scales: Python benefits from shorter code, Matlab from lower computational demands, and C++ shows reduced length, lines, and computational costs. Complexity patterns vary by language. Python and Matlab exhibit higher logical complexity, while C++ or Java are less affected. Structure also differs between languages. For Python and Matlab, which are more encapsulated languages, it is surprising that better code does not encourage function calls. For C++, contrary to the above two languages, some function calls can improve the accuracy. Overall, correct code preferences defy language conventions: High-abstraction languages prioritise procedural logic over fragmented encapsulation, while low-abstraction languages improve quality through modularity. Finally, Java seems to be little affected by the above metrics, and there is no significant difference between correct and incorrect code.

4.5 Different Models on MultiLingPoT

To further explore the applicability of MultiLingPoT, we repeat the method on other models. Since the selection strategy is independent of the model, we focus on the impact of different models on training. We choose three types of models: `DeepseekCoder-7B-v1.5` for other code models (Zhu et al., 2024), `CodeLlama-7B-Python-hf` for single-language code models (Roziere et al., 2023) and `Llama3-8B-Instruct` for non-code models (AI@Meta, 2024). Figure 4 shows the relative results and the Appendix D shows the specific values.

Among these three types of models, MultiLingPoT with self-consistency consistently preserves the best results, and the performance of each language is generally balanced, demonstrating the broad applicability of the MultiLingPoT in models with code capabilities. Certainly, there are still differences between the different models. For DeepseekCoder, both its SinglePoT and MultiLingPoT perform better, indicating that the performance of MultiLingPoT improves with the model's inherent capabilities. For Llama3, the performance is only matched by CodeLlama based on Llama2, indicating that the code capability of the model still affects the effectiveness of MultiLingPoT. Finally, CodeLlama-Python performs slightly better in Python, indicating that language-specific models can improve performance in their focus language without significantly affecting others.

5 Conclusion

In this paper, we explore the impact of multilanguage PoT in LLM mathematical reasoning during training and inference, and propose MultiLingPoT, an multilingual program reasoning method. For training, we construct multilingual PoT data to fine-tune the MultiLingPoT model, achieving a 2% performance improvement. For inference, we design multiple language selection strategies that include prior and posterior, and select the appropriate language to answer during inference, improving accuracy to 8% vs. the best unchosen result. Finally, we conduct analyses using software code metrics.

We find that key differences cross-languages manifest in encapsulation dependency and step-by-step logic. And controlled divergence from programming language conventions can improve computational performance.

Limitations

Our study is comprehensive but still has some limitations that we plan to address in future research. For the multilingual PoT data we construct, we use ChatGPT, as the work is conducted at that time. Although more advanced models, such as GPT-4 (OpenAI, 2023) and DeepSeek-R1 (Zheng et al., 2025), are now available and can generate higher-quality data, we do not rebuild the dataset due to time and budget constraints. In addition, there is room for improvement for selection strategies. Although we conduct many explorations, we still not find an prior selection method that is obviously effective. While posterior selection is effective, it is more computationally intensive for each problem. Therefore, it is still a challenge to investigate how to effectively mix different programming languages while minimizing the amount of computation required. For example, reinforcement learning can fully eliminate selection costs, while early-stop strategies or prior-posterior mixing are also worth considering. Furthermore, in code metric analysis, Python and Matlab lack inference training and have excessive function calls, so optimizing them is another interesting topic (Ma et al., 2025b). Finally, we currently focus only on the application of multiple programming languages to mathematical problems. Since PoT has been widely applied in various domains, we will also explore the use of multilingual PoT in other fields in the future.

Ethics Considerations

We hereby acknowledge that all authors of this work are aware of the provided ACL Code of Ethics and honor the code of conduct. In our work, we use publicly available data during the dataset construction process and perform secondary construction. We strictly follow the ChatGPT usage guidelines and performed subsequent validation of the generated content to minimise the risk of harmful content generation. During model training, we adopt a publicly standardised training process and used harmless datasets to safeguard the model.

References

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455.

AI@Meta. 2024. Llama 3 model card.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, et al. 2021. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74.

Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

GitHub. 2024. Octoverse: Ai leads python to top language as the number of global developers surges.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*.

Joy He-Yueya, Gabriel Poesia, Rose E Wang, and Noah D Goodman. 2023. Solving math word problems by combining language models with symbolic solvers. *arXiv preprint arXiv:2304.09102*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Zhanming Jie and Wei Lu. 2023. Leveraging training data in few-shot prompting for numerical reasoning. *arXiv preprint arXiv:2305.18170*.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. 2023. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. *arXiv preprint arXiv:2308.09583*.

Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Tegawende F Bissyande, Haoye Tian, and Bach Le. 2025. Unlocking llm repair capabilities in low-resource programming languages through cross-language translation and multi-agent refinement. *arXiv preprint arXiv:2503.22512*.

Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Libo Qin, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024. Python is not always the best choice: Embracing multilingual program of thoughts. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7185–7212.

Zhiyuan Ma, Zhenya Huang, Jiayu Liu, Minmao Wang, Hongke Zhao, and Xin Li. 2025a. Automated creation of reusable and diverse toolsets for enhancing llm reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24821–24830.

Zhiyuan Ma, Jiayu Liu, Xianzhen Luo, Zhenya Huang, Qingfu Zhu, and Wanxiang Che. 2025b. Advancing tool-augmented large language models via meta-verification and reflection learning. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 2078–2089.

Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. 2021. A diverse corpus for evaluating and developing english math word problem solvers. *arXiv preprint arXiv:2106.15772*.

Swaroop Mishra, Arindam Mitra, Neeraj Varshney, Bhavdeep Sachdeva, Peter Clark, Chitta Baral, and Ashwin Kalyan. 2022. Numglue: A suite of fundamental yet challenging mathematical reasoning tasks. *arXiv preprint arXiv:2204.05660*.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*.

Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197.

OpenAI. 2022. Chatgpt.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? *arXiv preprint arXiv:2103.07191*.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. *arXiv preprint arXiv:2305.14318*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Fabrizio Riguzzi. 1996. A survey of software metrics. *Università degli Studi di Bologna*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Konstantinos Skianis, Giannis Nikolentzos, and Michalis Vazirgiannis. 2024. Graph reasoning with large language models via pseudo-code prompting. *arXiv preprint arXiv:2409.17906*.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11888–11898.

Aline Lopes Timóteo, Alexandre Álvaro, Eduardo Santana De Almeida, and Silvio Romero de Lemos Meira. 2008. Software metrics: A survey. *Sl: sn*.

Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. 2024. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 56(12):1–41.

Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023a. One adapter for all programming languages? adapter tuning for code search and summarization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 5–16. IEEE.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023b. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. *arXiv preprint arXiv:2310.03731*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Kaiyuan Yang, Junfeng Wang, and Zihua Song. 2025. Raxcs: Towards cross-language code summarization with contrastive pre-training and retrieval augmentation. *Information and Software Technology*, page 107741.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2023. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*.

Yilun Zhao, Yitao Long, Hongjun Liu, Ryo Kamoi, Linyong Nan, Lyuhao Chen, Yixin Liu, Xiangru Tang, Rui Zhang, and Arman Cohan. 2024. Docmath-eval: Evaluating math reasoning capabilities of llms in understanding financial documents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16103–16120.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. 2025. A review on edge large language models: Design, execution, and applications. *ACM Computing Surveys*, 57(8):1–35.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Prompt Template

A.1 Prompt Template for Data Construction

The prompt template for ChatGPT to generate the multilingual PoT data is shown in List 1.

Listing 1: Instruction template for ChatGPT to generate multilingual PoT data.

```
Task Prompt:
Please use {program_type} functions to
solve math problems. The function name
is "solution()" and return the result.
The following are some cases:

Example Questions of GSM8K:
Question1: Natalia sold clips to 48 of
her friends in April, and then she sold
half as many clips in May. How many
clips did Natalia sell altogether in
April and May?

Question2: There are 381 pages in Elliot
 book.  He has already read 149 pages.
If he reads 20 pages a day for a week,
how many pages are still left to be read
?

Question3: Weng earns $12 an hour for
babysitting. Yesterday, she just did 50
minutes of babysitting. How much did she
 earn?

Question4: Alexis is applying for a new
job and bought a new set of business
clothes to wear to the interview. She
went to a department store with a budget
 of $200 and spent $30 on a button-up
shirt, $46 on suit pants, $38 on a suit
coat, $11 on socks, and $18 on a belt.
She also purchased a pair of shoes, but
lost the receipt for them. She has $16
left from her budget. How much did
Alexis pay for the shoes?

Example Solutions of Python in GSM8K:
def solution():
    clips_april = 48
    clips_may = clips_april / 2
    clips_total = clips_april +
    clips_may
    result = clips_total
    return result

def solution():
    pages_initial = 381
    pages_read = 149
    pages_per_day = 20
    num_days = 7  # 7 days in a week
    pages_read_in_week = pages_per_day *
     num_days
    pages_left = pages_initial -
    pages_read - pages_read_in_week
    result = pages_left
    return result

def solution():
    hourly_rate = 12
    minutes_worked = 50
    hours_worked = minutes_worked / 60
    earnings = hourly_rate *
    hours_worked
    result = earnings
    return result

def solution():
    budget = 200
    shirt = 30
    pants = 46
    coat = 38
    socks = 11
    belt = 18
    money_left = 16
    shoes = budget - (shirt + pants +
    coat + socks + belt + money_left)
    result = shoes
    return result

Example Solutions of C++ in GSM8K:
float solution() {
    float clips_april = 48;
    float clips_may = clips_april / 2;
    float clips_total = clips_april +
    clips_may;
    float result = clips_total;
    return result;
}

float solution() {
    float pages_initial = 381;
    float pages_read = 149;
    float pages_per_day = 20;
    float num_days = 7;  // 7 days in a
    week
    float pages_read_in_week =
    pages_per_day * num_days;
    float pages_left = pages_initial -
    pages_read - pages_read_in_week;
    float result = pages_left;
    return result;
}

float solution() {
    float hourly_rate = 12;
    float minutes_worked = 50;
    float hours_worked = minutes_worked
    / 60;
    float earnings = hourly_rate *
    hours_worked;
    float result = earnings;
    return result;
}

float solution() {
    float budget = 200;
    float shirt = 30;
    float pants = 46;
    float coat = 38;
    float socks = 11;
    float belt = 18;
    float money_left = 16;
    float shoes = budget - (shirt +
    pants + coat + socks + belt +
    money_left);
    float result = shoes;
    return result;
}

Example Solutions of Java in GSM8K:
```

```
public static double solution() {
    double clips_april = 48;
    double clips_may = clips_april / 2;
    double clips_total = clips_april +
    clips_may;
    double result = clips_total;
    return result;
}

public static double solution() {
    double pages_initial = 381;
    double pages_read = 149;
    double pages_per_day = 20;
    double num_days = 7;  // 7 days in a
     week
    double pages_read_in_week =
    pages_per_day * num_days;
    double pages_left = pages_initial -
    pages_read - pages_read_in_week;
    double result = pages_left;
    return result;
}

public static double solution() {
    double hourly_rate = 12;
    double minutes_worked = 50;
    double hours_worked = minutes_worked
     / 60;
    double earnings = hourly_rate *
    hours_worked;
    double result = earnings;
    return result;
}

public static double solution() {
    double budget = 200;
    double shirt = 30;
    double pants = 46;
    double coat = 38;
    double socks = 11;
    double belt = 18;
    double money_left = 16;
    double shoes = budget - (shirt +
    pants + coat + socks + belt +
    money_left);
    double result = shoes;
    return result;
}
```

Example Solutions of **Matlab** in **GSM8K**:

```
function result = solution()
    clipsApril = 48;
    clipsMay = clipsApril / 2;
    totalClips = clipsApril + clipsMay;
    result = totalClips;
end

function result = solution()
    totalPages = 381;
    pagesRead = 149;
    pagesPerDay = 20;
    daysInAWeek = 7;
    pagesInAWeek = pagesPerDay *
    daysInAWeek;
    remainingPages = totalPages -
    pagesRead - pagesInAWeek;
    result = remainingPages;
end

function result = solution()
    hourlyRate = 12;
    babysittingMinutes = 50;
    babysittingHours =
    babysittingMinutes / 60;
    earnings = hourlyRate *
    babysittingHours;
    result = earnings;
end

function result = solution()
    budget = 200;
    shirtCost = 30;
    pantsCost = 46;
    coatCost = 38;
    socksCost = 11;
    beltCost = 18;
    amountSpent = shirtCost + pantsCost
    + coatCost + socksCost + beltCost;
    remainingBudget = budget -
    amountSpent;
    shoesCost = budget - amountSpent;
    result = shoesCost;
end
```

Example Questions of **MATH**:

```
Question1: The function $f(x)$ satisfies
 [f(x + y) = f(x) f(y)]for all real
numbers $x$ and $y.$ If $f(2) = 3,$ find
 $f(6).$

Question2: Compute the sum of all the
roots of $(2x+3)(x-4)+(2x+3)(x-6)=0$.

Question3: A triangle in a Cartesian
coordinate plane has vertices (5, -2),
(10, 5) and (5, 5). How many square
units are in the area of the triangle?
Express your answer as a decimal to the
nearest tenth

Question4: How many nonnegative
solutions are there to the equation $x^2
 = -4x$?
```

Example Solutions of **Python** in **MATH**:

```
def solution():
    def f(x):
        if x == 2:
            return 3
        else:
            return f(2) * f(x - 2)
    result = f(6)
    return result

def solution():
    x = sp.symbols('x')
    equation = (2*x + 3)*(x - 4) + (2*x
    + 3)*(x - 6)
    roots = sp.solve(equation, x)
    result = sum(roots)
    return result

def solution():
    vertices = [(5, -2), (10, 5), (5, 5)
    ]
    area = 0.5 * abs((vertices[0][0]*(
    vertices[1][1]-vertices[2][1]) +
    vertices[1][0]*(vertices[2][1]-
    vertices[0][1]) + vertices[2][0]*(
    vertices[0][1]-vertices[1][1])))
```

```
    return round(area, 1)

import sympy as sp
def solution():
    x = sp.symbols('x')
    equation = x**2 + 4*x
    solutions = sp.solve(equation, x)
    non_negative_solutions = [sol for
    sol in solutions if sol >= 0]
    result = len(non_negative_solutions)
    return result
```

Example Solutions of C++ in MATH:

```
double f(double x) {
    if (x == 2) {
        return 3;
    }
    return f(2) * f(x - 2);
}
double solution() {
    return f(6);
}

double solution() {
    double root1 = -1.5;
    double root2 = 5;
    double sum_of_roots = root1 + root2;
    return sum_of_roots;
}

double solution() {
    double x1 = 5.0, y1 = -2.0;
    double x2 = 10.0, y2 = 5.0;
    double x3 = 5.0, y3 = 5.0;
    double area = 0.5 * std::abs(x1 * (
    y2 - y3) + x2 * (y3 - y1) + x3 * (y1
     - y2));
    return area;
}

int solution() {
    int a = 1;
    int b = 4;
    int discriminant = b * b - 4 * a *
    0;
    int root_count = 0;
    if (discriminant > 0) {
        double root1 = (-b + sqrt(
        discriminant)) / (2 * a);
        double root2 = (-b - sqrt(
        discriminant)) / (2 * a);
        if (root1 >= 0) root_count++;
        if (root2 >= 0) root_count++;
    } else if (discriminant == 0) {
        double root = -b / (2 * a);
        if (root >= 0) root_count++;
    }
    return root_count;
}
```

Example Solutions of Java in MATH:

```
public static double solution() {
    double a = Math.sqrt(3);
    double result = Math.pow(a, 6);
    return result;
}

public static double solution() {
    double a = 4;
    double b = -14;
    double c = -30;
    double discriminant = b * b - 4 * a
    * c;
    if (discriminant < 0) {
        return Double.NaN;
    } else {
        double root1 = (-b + Math.sqrt(
        discriminant)) / (2 * a);
        double root2 = (-b - Math.sqrt(
        discriminant)) / (2 * a);

        double sumOfRoots = root1 +
        root2;

        return sumOfRoots;
    }
}

public static double solution() {
    double x1 = 5;
    double y1 = -2;
    double x2 = 10;
    double y2 = 5;
    double x3 = 5;
    double y3 = 5;
    double area = 0.5 * Math.abs(x1 * y2
     + x2 * y3 + x3 * y1 - x1 * y3 - x2
    * y1 - x3 * y2);
    return area;
}

public static int solution() {
    int a = 1;
    int b = 4;
    int discriminant = b * b - 4 * a *
    0;
    int root_count = 0;
    if (discriminant > 0) {
        double root1 = (-b + Math.sqrt(
        discriminant)) / (2 * a);
        double root2 = (-b - Math.sqrt(
        discriminant)) / (2 * a);
        if (root1 >= 0) root_count++;
        if (root2 >= 0) root_count++;
    } else if (discriminant == 0) {
        double root = -b / (2.0 * a);
        if (root >= 0) root_count++;
    }
    return root_count;
}
```

Example Solutions of Matlab in MATH:

```
function result = solution()
    function value = f(x)
        if x == 2
            value = 3;
        else
            value = f(x - 2) * f(2);
        end
    end
    result = f(6);
end

function result = solution()
    syms x;
    equation = (2*x + 3)*(x - 4) + (2*x
    + 3)*(x - 6) == 0;
    roots_x = solve(equation, x);
    sum_of_roots = sum(roots_x);
    result = sum_of_roots;
```

```
end

function result = solution()
    x1 = 5; y1 = -2;
    x2 = 10; y2 = 5;
    x3 = 5; y3 = 5;
    area = 0.5 * abs(x1*(y2 - y3) + x2*(
    y3 - y1) + x3*(y1 - y2));
    result = area;
end

function result = solution()
    syms x
    equation = x^2 + 4*x;
    solutions = solve(equation);
    num_solutions = sum(double(solutions
     >= 0));
    result = num_solutions;
end
```

A.2 Prompt Template for Model Inference

The prompt template for MultiLingPoT to inference in different languages is shown in List 2.

Listing 2: Instruction template for MultiLingPoT to inference in different languages.

```
/* Instruction */
You are a math problem solving robot,
for the input math problem, write the
solution function to solve the problem
using the specified programming language
.

/* Input */
Write a {program_type} program to solve
the problem: {question}

/* Output in Python */
def solution():
    x, y = sp.symbols('x y')
    equation1 = x**2 - y
    equation2 = x + y - 1
    intersections = sp.solve((equation1,
     equation2), (x, y))
    distance = sp.sqrt((intersections
    [0][0] - intersections[1][0])**2 + (
    intersections[0][1] - intersections
    [1][1])**2)
    return distance

/* Output in C++ */
double solution() {
    // The equation of the first
    intersection is (0, 0)
    // The equation of the second
    intersection is (1 - sqrt(1 - 4*0),
    1 - sqrt(1 - 4*0))
    double x1 = 0;
    double y1 = 0;
    double x2 = 1 - sqrt(1 - 4 * 0);
    double y2 = 1 - sqrt(1 - 4 * 0);
    double distance = sqrt(pow(x2 - x1,
    2) + pow(y2 - y1, 2));
    return distance;
}

/* Output in Java */
public static double solution() {
    double x1 = 0;
    double y1 = 0;
    double x2 = 1 - Math.sqrt(1 - 4 * 0
    * 0);
    double y2 = 1 - 0;
    double distance = Math.sqrt(Math.pow
    (x2 - x1, 2) + Math.pow(y2 - y1, 2))
    ;
    return distance;
}

/* Output in Matlab */
function result = solution()
    syms x;
    y = x^2;
    equation1 = x + y == 1;
    equation2 = y == 0;
    solutions = solve([equation1,
    equation2], x);
    intersection_points = [double(
    solutions(1)), 0];
    distance = norm(intersection_points
    (1) - intersection_points(2));
    result = distance;
end
```

B Language Selection Implementation

The following are the implementation details of the language selection strategy in inference.

B.1 Prior Selection Strategy

The four specific implementation details of prior selection strategy are as follows.

Case-Based Choice Assuming that similar queries have similar solutions (Dong et al., 2022), the method selects languages based on their performance on training examples similar to the input query. For each input query, the method uses `text-embedding-3-small` (Neelakantan et al., 2022) to compute similarity and rank the relevant examples in the training set. Starting with the most similar examples, the method counts the number of correct answers for each programming language. The language that reaches 10 correct counts first is selected to answer the current query.

Small Model Scorer This method uses small parameter models as scorers for programming language selection. Given that our task involves both natural and programming languages, we choose `Bert-base-uncased` (Devlin, 2018) and `CodeBert-base` (Feng et al., 2020) as the base models. Specifically, this method uses the four programming languages' performance on the training set to train four scorers. Each scorer is responsible for evaluating a specific language. For an input query, the scorers provide a rating of [0,1]. The lan-

guage with the highest rating is chosen to answer the current query.

LLM Scorer Given that LLMs have better reasoning and understanding abilities, this method uses `Llama3-8B-Instruct` (AI@Meta, 2024) as a scorer for programming language selection. Similar to the small model scorer, this method trains a Llama3 scorer using the performance of four languages on the training set. Given the input query and chosen language, the scorer returns a "Yes" or "No" response. And we use the difference between the logprob of "Yes" and "No" as the score and select the highest scoring language to answer the current query.

Direct Perference Optimization This method performs DPO training (Rafailov et al., 2024) on the MultiLingPoT model, giving the model the ability to select preferences itself. Specifically, the method uses the performance of the four languages on the training set as the preference dataset. For the input query, the PoT of the language that answered correctly is chosen, while the PoT of the incorrect language is rejected. By DPO training on the preference dataset, the model can directly output the query-preferred language results without additional selection.

B.2 Posterior Selection Strategy

The four specific implementation details of posterior selection strategy are as follows.

Self Consistency Referring to the study by Wang et al. (2022), this method selects the final answer by voting on the PoT results in four languages. The answer with the most votes is chosen as the final answer. In case of a tie vote, one of the tied results is randomly selected as the final answer.

Small Model Scorer This method uses small parameter models as the scorers for programming language selection. All steps are the same as in the "Small Model Scorer" with the prior selection, except that both the query and the generated code are entered as criteria during training and inference. This allows the model to make selections not only based on the preferences from the query but also by considering additional factors, such as the completeness of the code, the library functions used, etc.

LLM Scorer This method uses LLM as a scorer for programming language selection. All steps are the same as in the "LLM Scorer" with the prior selection, except that both the query and the generated code are used as evaluation criteria.

Voting and Scoring Influenced by the fact that votes are often tied in complex dataset, we incorporate the self-consistency and LLM scorer methods. For the input query and the PoT results in the four languages, the method first votes on the PoT results. If there is a single highest-scoring answer, it is selected as the final output. Otherwise, the final answer is selected using the LLM scorer.

C Case Study

C.1 Correctness and Incorrectness in Different Languages

The following are some examples where Python/-Matlab are erroneous but C++/Java are correct.

Listing 3: Case Study of Correctness and Incorrectness in Different Languages.

```
Case 1
A line segment begins at (1, 3). It is 5
 units long and ends at the point (4, x)
 and x > 0. What is the value of x?
/* python/matlab */
def solution():
    x = sp.symbols('x')
    distance = sp.sqrt((4 - 1)**2 + (x -
     3)**2)
    equation = sp.Eq(distance, 5)
    solutions = sp.solve(equation, x)
    result = solutions[0]
    return result
/* c++/java */
public static double solution() {
    double x1 = 1;
    double y1 = 3;
    double length = 5;
    double x2 = 4;
    double y2 = y1 + Math.sqrt(length *
    length - (x2 - x1) * (x2 - x1));
    return y2; }

Case 2
What is the remainder when 2^{87} +3 is
divided by 7?
/* python/matlab */
function result = solution()
    remainder = mod(2^87 + 3, 7);
    result = remainder;
    end
/* c++/java */
public static int solution() {
    int base = 2;
    int exponent = 87;
    int modulus = 7;
    int result = 1;
    for (int i = 0; i < exponent; i++) {
        result = (result * base) %
        modulus; }
    result = (result + 3) % modulus;
    return result;}
```

Method	Language	Algebra	Count Prob.	Geom.	Int. Algebra	Num. Theory	Prealg.	Precalc.	Average
DeepseekCoder									
SinglePoT	Python-DA	49.40	37.97	26.01	27.18	61.03	62.96	27.36	41.70
MultiLingPoT	Python	47.95	31.64	25.37	23.92	60.26	60.81	26.87	39.54
	C++	53.95	41.56	27.07	22.74	52.20	64.39	26.39	41.18
	Java	48.77	43.67	29.42	23.79	54.70	62.72	26.87	41.42
	Matlab	35.21	43.45	26.86	14.50	51.82	56.15	24.45	36.06
	Self-Cons.	**58.23**	**49.57**	**30.49**	**28.36**	**62.95**	**68.21**	**30.02**	**46.83**
CodeLlama-Python									
SinglePoT	Python-DA	40.85	32.91	23.45	21.04	52.59	52.44	23.97	35.32
MultiLingPoT	Python	40.30	28.48	18.97	19.86	53.93	50.77	23.72	33.71
	C++	40.76	32.70	23.24	18.43	47.79	50.65	21.30	33.55
	Java	39.67	36.91	20.46	18.69	48.94	52.09	24.45	34.45
	Matlab	30.48	38.18	21.10	13.20	45.68	47.31	22.76	31.24
	Self-Cons.	**46.95**	**40.08**	**24.09**	**23.26**	**56.81**	**57.10**	**26.87**	**39.30**
Llama3									
SinglePoT	Python-DA	42.12	30.80	23.66	**24.44**	**55.85**	52.21	21.79	35.83
MultiLingPoT	Python	40.49	28.05	21.10	17.90	48.94	54.24	20.82	33.07
	C++	44.13	34.59	23.45	19.60	45.87	53.28	22.27	34.74
	Java	44.04	38.81	25.79	19.21	46.25	53.88	21.79	35.68
	Matlab	30.11	40.08	20.46	13.33	43.76	46.35	21.06	30.73
	Self-Cons.	**50.22**	**41.98**	**26.22**	23.92	54.70	**59.73**	**25.66**	**40.34**

Table 8: Results of different models using MultiLingPoT on complex datasets, including DeepseekCoder for the code model, CodeLlama-Python for the code model in a single language, and Llama3 for the non-code model.

There are two main reasons for this situation: (1) Incorrect use of library functions and (2) different solution logics. For the first type, Python/Matlab often encounter errors due to the excessive number of library functions, including calling non-existent library functions (hallucinations), failing to handle the output of library functions, incorrect parameter passing to library functions, etc. For the second category, we found that due to the abundance of library functions in Python/Matlab, they tend to perform direct calculations during problem-solving, while C++/Java are more likely to employ "clever solutions." This results in Python/Matlab consuming more computational resources in some problems and may even fail to provide answers due to exceeding computational limits. Similar to the second type, C++/Java are generally more sensitive to boundary conditions and exception handling issues.

C.2 Challenge of Language Selection

The following are some examples of language selection in multilingual PoT reasoning.

```
Case 1
Ravi has some coins. He has 2 more
quarters than nickels and 4 more dimes
than quarters. If he has 6 nickels, how
much money does he have?
Appropriate Language: C++

Case 2
There are 5 green candies, 3 blue
candies, and 4 red candies in a bag. If
Violet randomly picks a candy from the
bag, how likely is it that it's blue?
Appropriate Language: Python

Case 3
How many rows of Pascal's Triangle
contain the number 43?
Appropriate Language: Python

Case 4
Let C be the circle with equation x^2+12
y+57=-y^2-10x. If (a,b) is the center of
 C and r is its radius, what is the
value of a+b+r?
Appropriate Language: Java

Case 5
The integer x has 12 positive factors.
The numbers 12 and 15 are factors of x.
What is x?
Appropriate Language: Matlab

Case 6
```

Listing 4: Case Study of Challenge in Language Selection.

```
A telephone pole is supported by a steel
 cable which extends from the top of the
 pole to a point on the ground 3 meters
from its base. When Leah walks 2.5
meters from the base of the pole toward
the point where the cable is attached to
 the ground, her head just touches the
cable. Leah is 1.5 meters tall. How many
 meters tall is the pole?
Appropriate Language: C++
```

Among these, case 1 and case 2 are from gsm8k. For simple datasets, since the capabilities of different languages are not significantly different (basic arithmetic operations), the logical reasoning and judgment of conditions during inference can significantly impact the results, which makes language selection challenging. Case 3 to 6 are from MATH, which also poses difficulties in language selection. There are several reasons for this difficulty: (1) The problem is abstracted, so it may not be immediately clear what capabilities or libraries are required. (2) Even if a language has the corresponding library, it may not necessarily provide the correct solution. (3) Factors such as problem-solving logic and decoding randomness can also affect the results. Therefore, selecting the appropriate language, especially when relying solely on the query, is quite challenging.

D Different Models on MultiLingPoT

The specific results of different models training with multilingual PoT are shown in Table 8.