

# MATHCHAT: CONVERSE TO TACKLE CHALLENGING MATH PROBLEMS WITH LLM AGENTS

Yiran Wu<sup>1</sup>, Feiran Jia<sup>1</sup>, Shaokun Zhang<sup>1</sup>, Hangyu Li<sup>2</sup>, Erkang Zhu<sup>3</sup>, Yue Wang<sup>3</sup>,  
Yin Tat Lee<sup>4</sup>, Richard Peng<sup>5</sup>, Qingyun Wu<sup>1</sup>, Chi Wang<sup>3</sup>

<sup>1</sup>Pennsylvania State University <sup>2</sup>Imperial College London <sup>3</sup>Microsoft Research Redmond

<sup>4</sup>University of Washington <sup>5</sup>University of Waterloo

{yiran.wu, feiran.jia, shaokun.zhang, qingyun.wu}@psu.edu,  
{ekzhu, wang.yue, wang.chi}@microsoft.com, hl6021@ic.ac.uk,  
yintat@uw.edu, y5peng@uwaterloo.ca

## ABSTRACT

Employing Large Language Models (LLMs) to address mathematical problems is an intriguing research endeavor, considering the abundance of math problems expressed in natural language across numerous science and engineering fields. LLMs, with their generalized ability, are used as a foundation model to build AI agents for different tasks. In this paper, we study the effectiveness of utilizing LLM agents to solve math problems through conversations. We propose *MathChat*, a conversational problem-solving framework designed for math problems. *MathChat* consists of an LLM agent and a user proxy agent which is responsible for tool execution and additional guidance. This synergy facilitates a collaborative problem-solving process, where the agents engage in a dialogue to solve the problems. We perform evaluation on difficult high school competition problems from the MATH dataset. Utilizing Python, we show that *MathChat* can further improve previous tool-using prompting methods by 6%.

## 1 INTRODUCTION

With Large Language Models (LLMs) demonstrating remarkable proficiency in various tasks spanning diverse domains (Bubeck et al., 2023), they are deemed the potential foundation model for building autonomous agents (Xi et al., 2023; Wang et al., 2023a). Especially, multi-agent collaboration is a promising direction with the growing complexity of tasks being studied, with the benefit of information sharing and collective decision-making among different agents with specialized skills. It is compelling to explore the potential of LLMs in tackling mathematical problems considering the crucial role of mathematics (Wigner, 1990) and the prevalence of mathematical problems expressed in natural language throughout numerous scientific and engineering disciplines.

In this work, we investigate the potential of solving challenging math problems through conversations between agents. Due to the complex nature of these problems, we usually need to decompose them into multiple steps and can only make meaningful progress when all previous steps are correct. We believe conversations (together with code execution) are an ideal format, which enables iterative refining and debugging of each step. We propose *MathChat*, a conversational framework tailored to chat-based LLMs, where the math problem is solved with a mock conversation between an LLM-based agent and a user proxy agent (See Figure 1 for an example and Figure 2 for the workflow). We also study and incorporate effective prompting methods to instruct the LLM-based agent to solve challenging math problems more effectively.

We evaluate *MathChat* with GPT-4 on the MATH dataset (Hendrycks et al., 2021), a comprehensive collection of mathematical problems derived from various competitions and educational levels. We target the level-5 difficulty problems within this dataset, which primarily consist of challenging high school competition problems that even college students find difficult. Recognizing code execution as a major boost in performance, we compare two methods that both use Python: Program of Thoughts (PoT) prompt (Chen et al., 2022), and Program Synthesis prompt (Drori et al., 2022). We also include a vanilla prompt for reference. The evaluation shows that *MathChat* can further improve previous tool-using prompting methods by 6%, and it can reach 60% accuracy on half of

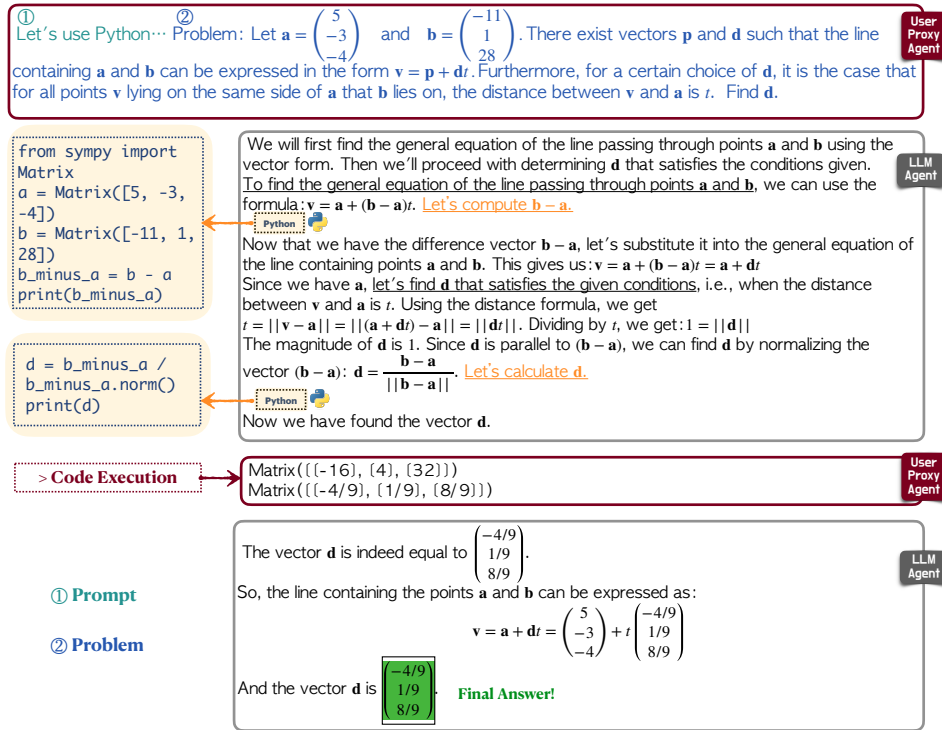


Figure 1: Example of a math problem-solving process with MathChat. The user proxy agent initiates a conversation by sending the math problem to be solved an LLM agent with preset prompt). From GPT-4’s response, the user proxy agent extracts all code and executes them sequentially. Valid code from previous runs is recorded and will be executed together with the new code to reflect the step-by-step reasoning progress of the model. The results will be returned to GPT-4 and GPT-4 will continue its problem-solving process. While GPT-4 solves this problem with only one turn of user message in this example, our framework allows multi-turn conversations and additional query handling, shown in Figure 3. The user proxy agent will do pattern-matching (in our case, the appearance of `\boxed{\}` containing a final answer) in the LLM agent’s response to determine whether to end the conversation.

the categories while having competitive performance across all categories. We also demonstrate the extensibility of MathChat with different prompts and different tools from our experiment. We conduct a detailed analysis of the failure reasons of all the methods evaluated.

## 2 RELATED WORK

**LLM Agent Systems** In the domain of LLM Agent Systems, various implementations have demonstrated the utility and diversity of multi-agent AI models. BabyAGI (BabyAGI, 2023) exemplifies an AI-powered task management system using multiple LLM-based agents with a static agent conversation pattern, while CAMEL (Li et al., 2023) showcases a communicative agent framework emphasizing role-playing and autonomous cooperation. Further, research on Multi-Agent Debate (Liang et al., 2023; Du et al., 2023) highlights the efficacy of agent debates in enhancing divergent thinking and factuality in LLMs. MetaGPT (Hong et al., 2023), a specialized application, demonstrates the use of GPTs in collaborative software development. AutoGen (Wu et al., 2023) is an open-source framework for creating diverse LLM applications with customizable, conversable agents using LLMs, human input, and tools.

**Prompting Methods** Creative ways of using LLMs to solve math problems have emerged lately (Wang et al., 2022; Zhou et al., 2023; Zheng et al., 2023; Chen et al., 2021; Weng et al., 2022). One particular endeavor is using LLMs to offload arithmetic calculations and other basic operations involved in math problem-solving to programs (Drori et al., 2022; Chen et al., 2022; Gao et al., 2022). Cumulative Reasoning (Zhang et al., 2023) decomposes tasks into smaller com-

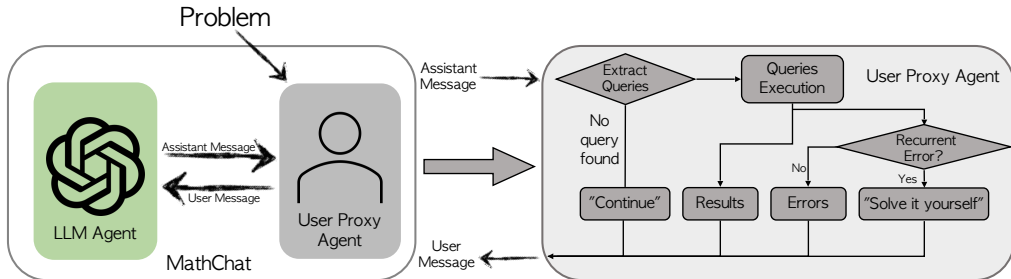


Figure 2: MathChat workflow: After a math problem is fed into MathChat, the user proxy agent will initiate a conversation with the LLM agent to solve the problem. In each turn of interaction, the user proxy agent processes the message from the LLM agent (Assistant Message) and responds with a User Message. This process continues until the user proxy agent detects a certain pattern to end the conversation. The process in the rectangular on the right-hand side of the figure shows the inner workflow of the user proxy agent once an Assistant Message is received. It shows the functionality of executing any tool-using queries, such as Python code. It is also responsible for giving different instructions corresponding to different types of messages from the LLM agent (More in Appendix A). To illustrate this feature of the proxy agent, we give a concrete example in Figure 3.

ponents, streamlines the solving process, and generates thoughts in a cumulative manner. Plan & Solve prompting (Wang et al., 2023b) ask the LLM to first generate a plan and then solve it accordingly. Other general methods used to improve reasoning can also be applied to math problems: (1) Chain-of-thought (CoT) prompting (Wei et al., 2022; Kojima et al., 2022) elicits step-by-step reasoning process from LLMs. (2) Another effective way is to prompt LLMs to solve problems in a multi-stage manner (Dua et al., 2022; Press et al., 2022; Creswell et al., 2022; Long, 2023; Paranjape et al., 2023; Yao et al., 2022; Yang et al., 2022; Long, 2023; Besta et al., 2023). Least-to-most prompting (Zhou et al., 2022) and Decomposed prompting (Khot et al., 2022) break down a complex problem into smaller subproblems, and the subproblems are solved sequentially to reach the final solution. (3) Utilizing tools can significantly boost the performance of LLMs (Shen et al., 2024; Parisi et al., 2022). ReAct (Yao et al., 2022) and ART (Paranjape et al., 2023) both use few-shot prompting to interleave step-by-step reasoning and tool-using. (4) Self-consistency (Wang et al., 2022), built on top of CoT, samples several different reasoning paths for a problem and selects the answer with the majority vote. Li et al. (2022) extends self-consistency by training a verifier to verify the correctness of each step. By decomposing a problem-solving process, Tree-of-Thoughts (Yao et al., 2023) proposes a set of thoughts for each intermediate step, exploring and maintaining the most promising thoughts for sequential actions.

### 3 MATHCHAT: A CONVERSATIONAL FRAMEWORK FOR MATH PROBLEM SOLVING

In this section, we introduce MathChat, a conversational framework for math problem-solving.

**A conversational framework with user proxy agent.** MathChat is a framework that simulates a mock conversation between an LLM agent (GPT-4 in our case) and a user proxy agent. Here a user proxy agent is an agent playing the user’s role in conversations with the LLM agent. In MathChat, the LLM agent and the user proxy agent work together to solve the math problem. The workflow of this framework is presented in Figure 2. The user proxy agent takes a math problem to be solved as input and would initiate a conversation with the LLM agent. The initial message from the user proxy agent consists of an initial prompt and the problem to be solved. The initial prompt is used to instruct the LLM agent to solve the problem collaboratively with the user (effectively the user proxy agent in the MathChat system) in a certain desired manner. This framework is designed in this conversational manner in order to leverage the chat-optimized feature of state-of-the-art LLMs, e.g., GPT-4. Another distinct benefit of this framework is that it enables multi-turn dialogues, which can be particularly useful in addressing complex issues that require multi-step reasoning and tool using.

**Prompting and tool-using in MathChat.** With proper modifications, effective prompting methods from existing research, such as CoT and tool-using, can be integrated into the MathChat framework. Specifically, for the prompt in the initial message, we aggregate multiple effective prompting

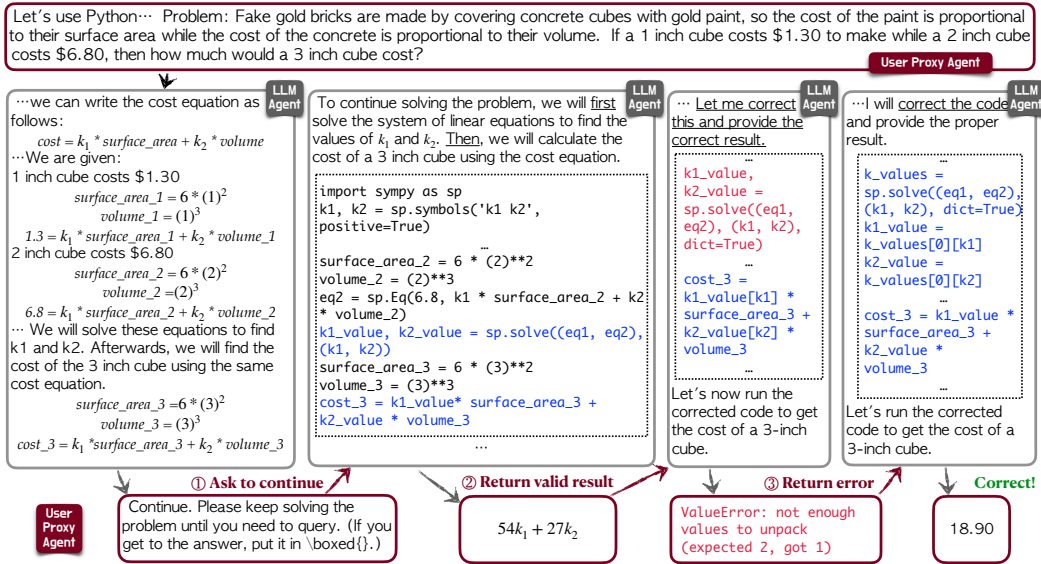


Figure 3: An example demonstrating how the user proxy agent handles different types of messages received from GPT-4 in MathChat. Specifically, the user proxy agent may respond in the following ways: (1) asking the LLM agent to continue because no code block (i.e., query) is detected; (2) returning the valid results from code execution; and (3) returning the error message from Python execution. Note that GPT-4 may change the query if the old code is undesired based on the message from the user proxy agent. In the last step, GPT-4 corrects the query, and the final result is returned.

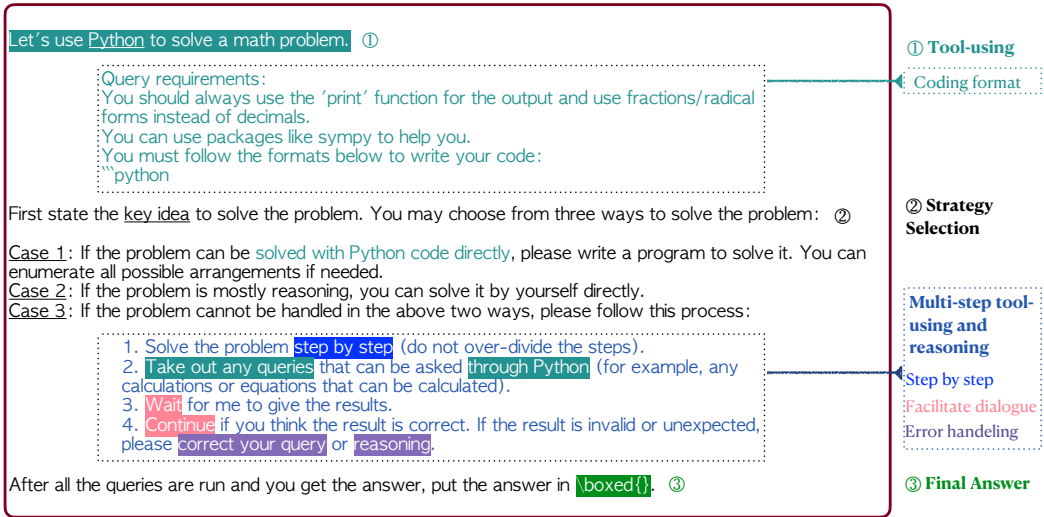


Figure 4: The prompt used in the initial message of the user proxy agent in MathChat. It instructs the LLM agent to solve a problem collaboratively with the user proxy agent in a certain way.

techniques to instruct the LLM agent. We present the designed prompt in Figure 4, which consists of three main components.

- Tool-using Prompt: This component prompts the LLM to use Python programming in the correct format to tackle the problem. We use the 'query requirement' subsection to specify the coding format so that the user proxy agent can parse the code and return the corresponding results.
- Problem-Solving Strategy Selection Prompt: This component instructs the LLM agent to select from three possible problem-solving strategies and to perform multi-stage reasoning and tool-

using in the last strategy. The problem-solving strategies include the following three cases, which cover the most effective strategies from existing literature on math problem-solving. (*Case 1*) *Write a Python program to solve the problem directly.* This corresponds to single-stage tool-using methods similar to Gao et al. (2022); Drori et al. (2022); Chen et al. (2022). (*Case 2*) *Solve the problem directly without Python.* This strategy allows GPT-4 to exercise its inherent reasoning capacity to solve the problem at hand. (*Case 3*) *Solve the problem step by step and use Python to help with math operations.* If the first two ways are not suitable, we ask the model to choose this way to solve the problem. We craft a zero-shot version of the multi-step tool-using prompt that allows the model to flexibly interleave between multi-step reasoning and Python code, similar to Yao et al. (2022); Paranjape et al. (2023); Schick et al. (2023). In this case, we also ask the model to handle errors and unexpected results from the run of programs Ni et al. (2023).

- **Final Answer Encapsulation Prompt:** This component of the prompt instructs the LLM agent to enclose the final answer in `\boxed{}`, which will be used as an indicator to end the conversation. This interaction between the LLM agent and the user proxy agent will not be ended until `\boxed{}` is detected or max rounds of conversations are reached.

We acknowledge that there could be alternative ways to design the prompt. Fortunately, it is fairly easy to refine the prompt, for example, further enabling the usage of Wolfram Alpha in addition to Python, in our framework. We perform an empirical evaluation accordingly to test two alternative versions of the prompt in Section 5.2.

## 4 EVALUATION

**Dataset.** We perform evaluations on all the level-5 (the highest difficulty level) problems from the test set of MATH dataset Hendrycks et al. (2021). Compared to other datasets for mathematical problems such as GSM8k Cobbe et al. (2021), the level-5 problems are much more challenging and include the application of theorems and complex equation derivation. The MATH dataset has 7 categories of problems: Prealgebra, Algebra, Number Theory, Counting and Probability, Geometry, Intermediate Algebra, and Precalculus. In our evaluation, we remove Geometry from the evaluation to make it consistent with previous work Drori et al. (2022) (additional explanation in Appendix B).

**Evaluated Methods.** Most previous work uses few-shot examples to elicit the reasoning of LLMs and tool-using. It is important to select similar examples to the unanswered problem, and then annotate the examples to cover all the cases that the LLMs might encounter. A considerable amount of effort and careful consideration are required in this process. For example, Khot et al. (2022); Zhou et al. (2022) relies on elaborate examples to showcase the patterns, Paranjape et al. (2023) maintains an example library to choose examples. Note that these methods use elementary math problems and it requires even more effort to prepare and choose the examples needed for challenging math problems. On the other hand, multiple existing studies OpenAI (2023); Bubeck et al. (2023) reveal GPT-4’s remarkable capacity to follow instructions. Thus, we are interested in zero-shot prompting techniques that could enhance math-solving of GPT-4, without any example selection and annotations. Following this criterion, we evaluate our `MathChat` framework with the introduced prompt and the following methods which are all zero-shot methods: vanilla prompt, Program of Thoughts Chen et al. (2022), and the Program Synthesis prompt from Drori et al. (2022).

1. **Vanilla prompting:** GPT-4 can perform CoT reasoning without few-shot examples. To evaluate GPT-4’s performance on solving the problem directly, we use a default prompt adapted from the few-shot prompt in MATH dataset: "Solve the problem carefully. Put the final answer in `\boxed{}`. {Problem}".
2. **Program of Thoughts (PoT):** We use the zero-shot PoT prompting from Chen et al. (2022), which asks a model to write a `Solver` function to solve a problem and return the final answer directly.
3. **Program Synthesis (PS) prompting:** Similar to PoT, the Program Synthesis (PS) prompting method Drori et al. (2022) uses a prompt to ask the model to write a program to solve a problem: "Write a program that answers the following question: {Problem}"

**Evaluation Details.** We compare different methods on GPT-4 with the default configurations from the OpenAI API. In `MathChat`, we allow a max round of 15 messages between GPT-4 and the user proxy agent. The agent will explicitly ask GPT-4 to solve each step by itself if it detects errors from 3 consecutive executions. To avoid extremely long responses from the user proxy agent, the agent

will replace any result that exceeds 600 tokens with a warning text in the user message to ask the GPT-4 to revise the previous code. We manually go through the answer of all the methods to count all the correct answers. For vanilla prompt, Program Synthesis, and MathChat, we ask GPT-4 to enclose the final answer in `\boxed{\}`, so only the answers in the box will be extracted. For PoT, we follow the original paper to take the return of the `Solver` function as the final answer.

## 5 RESULTS

### 5.1 MAIN RESULTS

We perform an evaluation on six categories of level-5 problems from the MATH dataset. We report the problem-solving accuracy of different methods in each category in Table 1. Compared to vanilla prompting, which shows the native capability of GPT-4, using Python with PoT or PS improves the overall accuracy by around 10%. We can see this improvement mostly in the categories that involve more number manipulations (Counting & Probability and Number Theory) and more challenging categories (Intermediate Algebra and Precalculus). For Algebra and Prealgebra, however, PoT and PS have little improvement or even lead to lower accuracy. Compared with PoT and PS, MathChat can further improve the total accuracy by around 6%, and have competitive performance across all the categories. It is worth highlighting that MathChat improves the accuracy in the Algebra category over other methods by around 15%. Considering all the methods, Intermediate Algebra and Precalculus can only be solved with a low accuracy rate of around 20%. More than half of the problems from the other categories can be solved correctly by MathChat.

Problem Count	Algebra	C.Prob	I.Alg	N.Theory	Prealg	Precalc	Total
	307	123	280	154	193	135	1192
<b>MathChat</b>	<b>59.93%</b>	<b>52.03%</b>	17.85%	60.39%	<b>60.10%</b>	<b>19.26%</b>	<b>44.71%</b>
PoT	42.67%	50.41%	17.50%	54.55%	52.33%	16.30%	37.67%
PS	43.32%	44.71%	<b>20.36%</b>	<b>61.03%</b>	55.96%	18.52%	39.60%
Vanilla	46.58%	25.20%	2.86%	28.57%	54.92%	7.41%	28.69%

Table 1: Accuracy on all the problems with difficulty level-5 from different categories of the MATH dataset with different methods.

### 5.2 ADDITIONAL EVALUATION ON MATHCHAT WITH ALTERNATIVE PROMPTS

Problem Count	Algebra	C.Prob	I.Alg	N.Theory	Prealg	Precalc	Total
	50	50	50	50	50	50	300
<b>MathChat w/ Two-tools</b>	<b>33</b>	22	6	27	29	10	127
<b>MathChat w/ Python</b>	26	19	7	22	<b>31</b>	<b>13</b>	118
<b>MathChat</b>	30	<b>24</b>	8	<b>34</b>	28	10	<b>134</b>
PoT	20	19	9	24	24	7	103
PS	17	19	<b>12</b>	31	26	5	110
Vanilla	26	13	1	17	21	1	79

Table 2: Additional evaluation of MathChat with two alternative prompts. 50 problems are sampled from each problem category for this evaluation. MathChat w/Two-tools and MathChat w/ Python are two alternative prompts.

MathChat allows easy incorporation of different prompts and tools. We perform an additional evaluation to test two alternative initial prompts with MathChat to demonstrate its extensibility. (1) A simplified prompt with Python: In this alternative, we only keep the ‘query requirements’ subsection for python coding format and the **step-by-step tool-using** (i.e., case 3) from the default prompt. (2) A simplified prompt with Python and Wolfram Alpha: In this alternative, on top of alternative (1), we add Wolfram Alpha, a computational engine, as an additional tool for the LLM agent to choose from. Details of these two alternative prompts are in Appendix B. We perform an evaluation on randomly sampled 50 examples from each of the six problem categories. We also include results from other methods on the sample problems for comparison in Table 2. MathChat

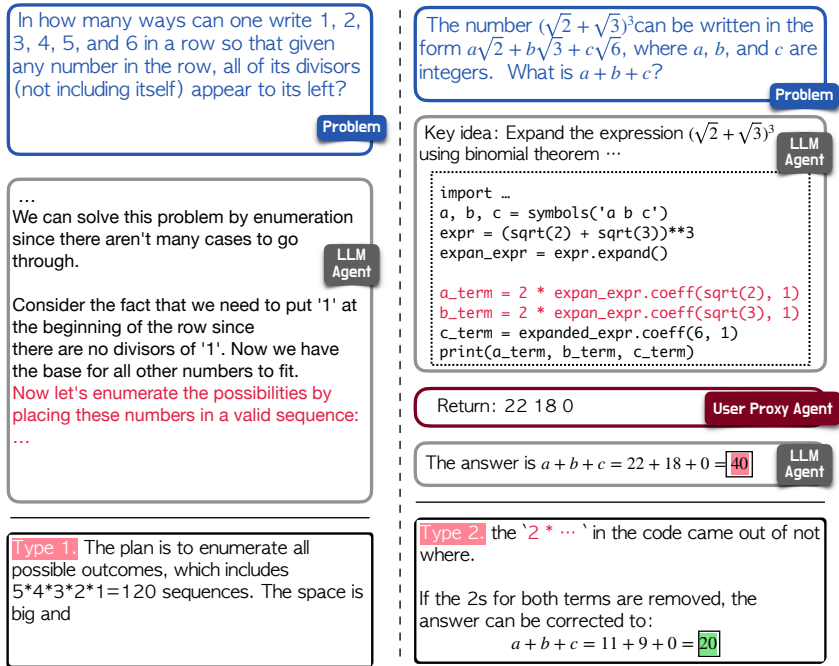


Figure 5: One example is selected for each of the first two failures. **Type 1 failure:** in the first problem, the LLM agent fails to give a plausible plan. It chooses to enumerate all sequences, and it does not use tools to help with it. **Type 2 failure:** the second problem shows that the model fails to give the correct code to solve the problem, while it follows the problem requirements and the overall direction is correct. With minor changes to the code, the final answer can be correct.

still performs better than other methods with the two newly crafted prompts. With MathChat, the step-by-step prompt that allows both Python and Wolfram performs the best on Algebra, while the new prompt with only Python solves the most problems on Prealgebra and Precalculus, but has a worse performance on Number Theories. Overall, MathChat with the default prompt still performs the best.

## 6 FAILURE ANALYSIS

### 6.1 FAILURE REASONS

We first summarize the failure cases according to the reasons for failure, based on the systematic math problem-solving process established by George Pólya Polya (2004). The process consists of (1) understanding the problem; (2) devising a plan; (3) executing the plan; (4) reviewing and extending. We observe failures of the following three main types. We give one example each for the two types of failures in Figure 5. More example are provided in Appendix D.

**Type 1. Failure to devise or select an appropriate plan or path to solve the problem.** This type encompasses cases where GPT-4 fails to provide a proper way to approach the problem. In these instances, the answer is consistently incorrect, even though each individual step of the calculation is accurate. Failure cases of this type are typically tricky problems that even math experts find challenging.

**Type 2. Failure to flawlessly execute the devised plan.** Math problem-solving requires rigorous and precise reasoning steps. A minor error in calculation or symbol manipulation could be fatal and lead to a wrong answer. This type of error is considered ‘minor’ because they are easier to be fixed. This type of error contributes to a fair amount of failures, where the overall direction of the problem-solving is correct, but one mistake in a basic derivation leads to the wrong answer. Note that an error for Python execution is also included in this type, where GPT-4 fails to write a runnable code leading to the error.

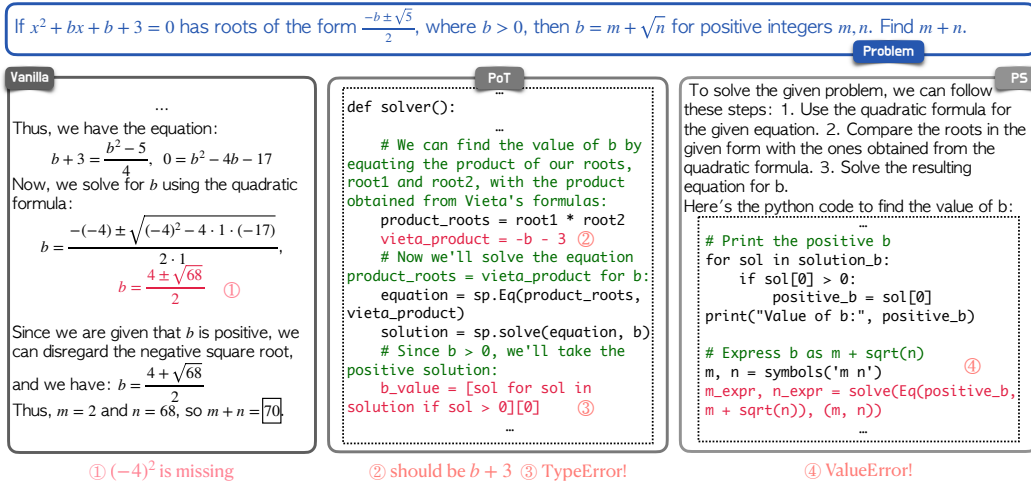


Figure 6: An example where MathChat is correct and others fail. All other methods fail due to Type 2 failure. 1. Vanilla prompt: when calculating  $b$ , didn't include  $-4^2$ . 2. PoT: it first calculates `vieta_product` wrong, even if this is corrected, another `TypeError` will occur. 3. PS: it solves for  $b$  correctly, but gets an `ValueError` when using the program to extract  $m$  and  $n$ .

**Type 3. Other technical errors.** There are other technical errors causing the failure. One example of such failure is lack of information due to the removal of ASY code.

### 6.2 FAILURE CASES USING DIFFERENT METHODS ON GPT-4

In Table 3, we present the frequency of successful outcomes for each method (represented in each row), while all other methods fail, categorized according to different problem instances. This table serves to highlight the distinct advantage that a particular method exhibits across various problem categories. Similarly, in Table 4, we summarize the frequency of instances where one method fails while all other methods succeed. A high number in this table signifies the unique disadvantage of the method in question.

These statistics demonstrate the robustness of MathChat in comparison to other methods. MathChat leverages conversation to enhance error correction when utilizing external tools, thereby hypothesizing a reduction in failures within the third type.

	Algebra	C.Prob	I.Alg	N.Theory	Prealg	Precalc	Total
MathChat	<b>27</b>	8	21	<b>13</b>	6	<b>9</b>	<b>84</b>
PoT	11	<b>9</b>	19	6	3	5	53
PS	12	6	<b>22</b>	11	<b>10</b>	8	69
Vanilla	12	4	5	3	<b>10</b>	3	37

Table 3: The number of problems where one method succeeds, and all the other methods fail (the higher the better for the concerned method in each row).

	Algebra	C.Prob	I.Alg	N.Theory	Prealg	Precalc	Total
MathChat	<b>6</b>	<b>2</b>	<b>0</b>	<b>5</b>	<b>4</b>	<b>1</b>	<b>18</b>
PoT	22	5	0	6	18	2	53
PS	17	5	1	<b>5</b>	14	<b>0</b>	42
Vanilla	16	19	11	28	19	5	98

Table 4: The number of problems where one method fails and all the other methods succeed (the lower, the better for the concerned method in each row).



We take one example from each table to analyze the failures of these methods. We first take an Algebra problem that `MathChat` succeeds but others fail to analyze the failures of other methods (Figure 6) For this problem, other methods fail to execute this plan without any mistakes, causing the second type of failure. While vanilla prompting has a calculation error, the other two methods get execution errors from running the code. We run these methods three more times and they still fail to solve the problem. From Table 4, we take the only Precalculus instance that `MathChat` is wrong while all the other methods are correct. Through investigating, we find that `MathChat` gives longer solutions to the problem than all the other methods, and also contains Type 2 failures. This problem might indicate a potential correlation between the accuracy and length of responses. We present more details in investigating this possible correlation and also the Precalculus example in Appendix D.

## 7 SUMMARY AND FUTURE WORK

### 7.1 SUMMARY

In this paper, we introduce `MathChat`, a conversational framework to solve math problems with the collaboration of an LLM agent and a user proxy agent. `MathChat` is designed for chat-optimized models like GPT-4, and it is extensible to be used with different prompts and different tools with minimal effort. Based on the framework, we also derive a prompt that aggregates previous prompting techniques to be used on `MathChat`. Our evaluation of level-5 problems from the MATH dataset demonstrates the effectiveness of `MathChat` to solve more complex and challenging problems. Despite its improvements over previous methods, the results show that complex math problems is still challenging for recent powerful LLMs, like GPT-4, even with help from external tools. We discuss potential directions to further improve math problem-solving below.

### 7.2 ENHANCED AGENT SPECIALIZATION IN PROBLEM SOLVING

*The Society of Mind* Minsky (1988) posits that intelligence emerges from the interaction of relatively simple agents. In `MathChat`, where two agents collaborate to solve math problems, the behavior of the LLM agent, guided by specific instructions for task completion and decision-making, shows significant variance. To enhance consistency and effectiveness, it could be beneficial to decompose this process into specialized tasks, each handled by a dedicated agent. One agent, for instance, could focus on comprehending and developing initial solutions, while another evaluates the most suitable problem-solving strategy for the given problem.

Other than decomposing the solving process, it is possible to categorize problems by type, difficulty level, or other aspects, and accordingly select the most effective agent (prompting method) for each category. Our analysis in Section 6.2 demonstrates that various methods exhibit distinct advantages depending on the problem type. This approach is akin to the Mixture of Experts model Shazeer et al. (2017), where specific prompting strategies are used in place of sub-neural networks, and it also aligns with the concept of prompting chaining Wu et al. (2022), which involves classifying a task under various scenarios for targeted resolution.

### 7.3 ASSISTANCE IN HUMAN PROBLEM-SOLVING

While LLMs is showing great potential to aid in human problem-solving, we recognize that much work remains in developing a reliable LLM-based problem-solving assistant. When conducting failure analysis in Section 6, we can spot calculation errors in LLM responses easily but may struggle with identifying logical or factual inaccuracies, especially with unfamiliar concepts. This could lead to potential misinformation, especially for students who are learning new concepts and have weaker judgements. Although our evaluation indicates that incorporating Python enhances LLM problem-solving abilities, relying solely on Python or LLMs has limitations, as Python solutions (such as brute-force or simulations) may not suit human learning needs.

A possible mitigation would be to verify each step of the solving process with external tools and established knowledge. When the LLM generates each intermediate step to solve a problem, Python can be used to check for calculation errors in the step, and external databases can be consulted to validate any theorems mentioned.

## REFERENCES

- BabyAGI. Github — babyagi. <https://github.com/yoheinakajima/babyagi>, 2023.
- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023.
- Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Antonia Creswell, Murray Shanahan, and Irina Higgins. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2205.09712*, 2022.
- Iddo Drori, Sarah Zhang, Reece Shuttlesworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, et al. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.
- Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.
- Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. *arXiv preprint arXiv:2212.04092*, 2022.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.
- Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.
- Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.
- Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large scale language model society, 2023.
- Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. On the advance of making language models better reasoners. *arXiv preprint arXiv:2206.02336*, 2022.

- Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. Encouraging divergent thinking in large language models through multi-agent debate, 2023.
- Jieyi Long. Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291*, 2023.
- Marvin Minsky. *Society of mind*. Simon and Schuster, 1988.
- Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. *arXiv preprint arXiv:2302.08468*, 2023.
- OpenAI. Gpt-4 technical report, 2023.
- Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.
- Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
- George Polya. *How to solve it: A new aspect of mathematical method*. Number 246. Princeton university press, 2004.
- Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.
- Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.
- Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugging-gpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36, 2024.
- Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023a.
- Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023b.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.
- Yixuan Weng, Minjun Zhu, Shizhu He, Kang Liu, and Jun Zhao. Large language models are reasoners with self-verification. *arXiv preprint arXiv:2212.09561*, 2022.
- Eugene P Wigner. The unreasonable effectiveness of mathematics in the natural sciences. In *Mathematics and science*, pp. 291–306. World Scientific, 1990.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

- Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pp. 1–10, 2022.
- Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.
- Jingfeng Yang, Haoming Jiang, Qingyu Yin, Danqing Zhang, Bing Yin, and Diyi Yang. Seqzero: Few-shot compositional semantic parsing with sequential prompts and zero-shot models. *arXiv preprint arXiv:2205.07381*, 2022.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.
- Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. Cumulative reasoning with large language models. *arXiv preprint arXiv:2308.04371*, 2023.
- Chuangyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. Progressive-hint prompting improves reasoning in large language models. *arXiv preprint arXiv:2304.09797*, 2023.
- Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023.
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

## A SUPPLEMENTARY DETAILS ON THE USER PROXY AGENT

The user proxy agent in `MathChat` takes a problem and put it in a message with an initial prompt, and sends the message to the LLM agent. Then the agent is responsible for extracting and executing queries and also providing additional guidance. Here are all functionalities of the user proxy agent (the workflow is shown in Figure 2):

1. **Extract Queries:** The user proxy agent needs to match the pattern specified in the initial message to extract all tool-using queries. With our designed prompt, the agent matches all code blocks in the message and extracts the code.
2. **"Continue":** If no query is detected in the message, the agent will send this message to the LLM agent: `"Continue. Please keep solving the problem until you need to query. (If you get to the answer, put it in \boxed{ }".` This asks the agent to keep solving the problem and reminds it to end the conversation by putting the answer in the box.
3. **Query Execution:** Any tool-using queries extracted will be executed sequentially. For Python, we set the time limit to be 5 seconds for execution. As shown in Figure 1, the previous valid code is recorded. All the execution results will be concatenated sequentially (including errors).
4. **Recurrent Error detection:** If LLM agent sends 3 consecutive errors, the user proxy agent will replace the third error message with this message: `"Please revisit the problem statement and your reasoning. If you think this step is correct, solve it yourself and continue the next step. Otherwise, correct this step."`. To avoid sticking to this error, the LLM agent is asked to solve this step without tools and move on.
5. **Repetitive results:** This is not shown in the workflow, but the agent also detects another situation where the LLM agent gives the same tool-using query from the last one or the result is the same as the last query. Then the message is appended to the execution result to remind the agent to avoid giving the same queries: `"Your query or result is same from the last, please try a new approach."`.
6. **Long query results:** It is possible that LLM agent requires a query result that is too long to be passed back (such as long results from the print function in a for loop in Python). The proxy agent will replace any query result that is longer than 2000 chars (approximately 600 tokens) with this message: `"Your requested query response is too long. You might have made a mistake. Please revise your reasoning and query."`.

In `MathChat`, if the tool-using query and the end indicator are detected in the same message, the result from the query will be returned, and the conversation will continue. This is to prevent early stops where the LLM agent predicts the execution result and puts it in a box other than waiting for the result.

## B SUPPLEMENTARY DETAILS ON EXPERIMENT SETTINGS

Rational in removing the geometry problems from testing: Most geometry problems from this dataset contain an `Asymptote` code to plot the figure. But the currently available version of GPT-4 cannot accept image input. If the raw code is included, it can leak information to the model through exact numbers for the coordinates. Taking these issues into consideration, we skip the evaluation on Geometry problems and remove `ASY` code from all the other categories (though this could result in a lack of enough information for some problems). The correct answer to each problem is deterministic and is enclosed in `\boxed{ }` in the dataset as ground truth (but not disclosed to the methods solving the problem).

The code is in this GitHub repository. In our experiment, we use the default configuration from OpenAI, specifically `temperature=1`, and `max_token=inf` (See OpenAI API Reference for more details). We use the system message "You are a helpful assistant" for vanilla prompt, PS, and `MathChat`. For PoT, we do not add this system message, since our evaluation shows that PoT

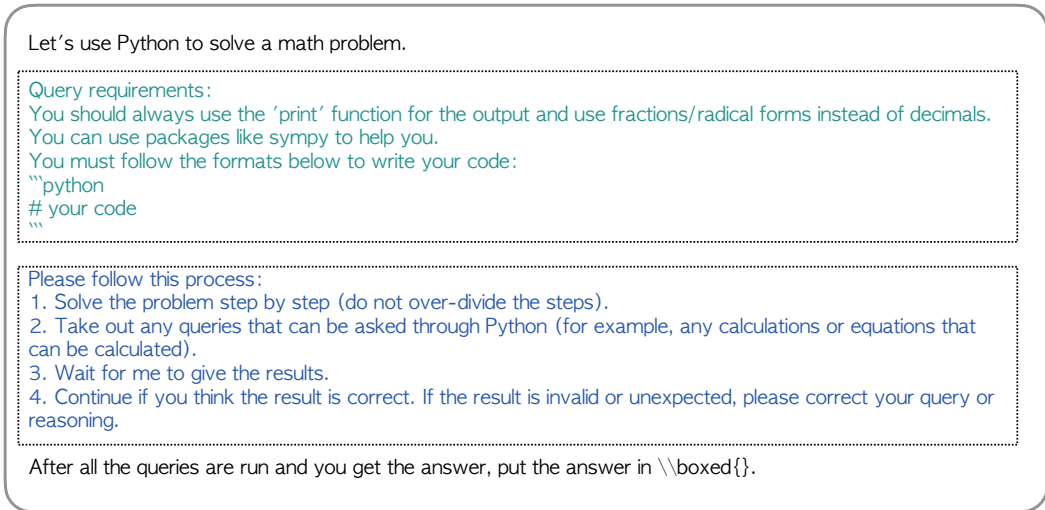


Figure 7: The Python prompt used on MathChat from Section 5.2.

without system message has a better performance. We discuss the effect of system message below in Section C.

Here is the prompts for PoT Chen et al. (2022), PS Drori et al. (2022), and the additional two prompts we designed:

- **Program of Thoughts (PoT)**. See Figure 10. The whole prompt uses the Python code format, where information such as the problem or instructions is in the comments.
- **Program Synthesis (PS)**. The prompt for PS is "Write a program that answers the following question: {Problem}". Since the definition of "program" is unclear and sometimes the LLM agent won't write code to solve the problem, we add the keyword 'Python' in front of 'program'. After the message is returned, we used the proxy agent to return the result (by default, GPT-4 would return the code in the code block). Then we send another message to the model with the Python execution result and ask it to enclose the final answer: {Return from Python}. Please put the final answer in `\boxed{}`. See Figure 9 for an example of the whole process.
- **Python prompt (w/ MathChat)**. See Figure 7.
- **Two-tools prompt (w/ MathChat)**. See Figure 8.

## C SUPPLEMENTARY EXPERIMENTS AND RESULTS

We further evaluate a vanilla few-shot prompt, PoT with and without system message, and Vanilla prompt with and without system message on the randomly selected 50 problems from each category and present the results in Figure 5.

In the few-shot prompt, we randomly select 3 level-5 problem-solution pairs from the train set. These examples are selected from each category and are used for all the problems from that category. The vanilla few-shot prompt starts with "Solve the problem carefully. Put the final answer in `\boxed{}` just like the vanilla prompt, and then three "Problem: ... Solution: ..." pairs are attached. Compared with the vanilla prompt, adding three additional examples does not make any obvious difference, and the overall performance is slightly worse.

From our experiment, we also notice that the system message affects the performance of the LLM agent. However, the impact significantly differs between methods. As shown in Table 5, using a system message is crucial for the Vanilla prompt: adding the system message doubles the success rate compared to the one without a system message. However, for PoT, adding the system message

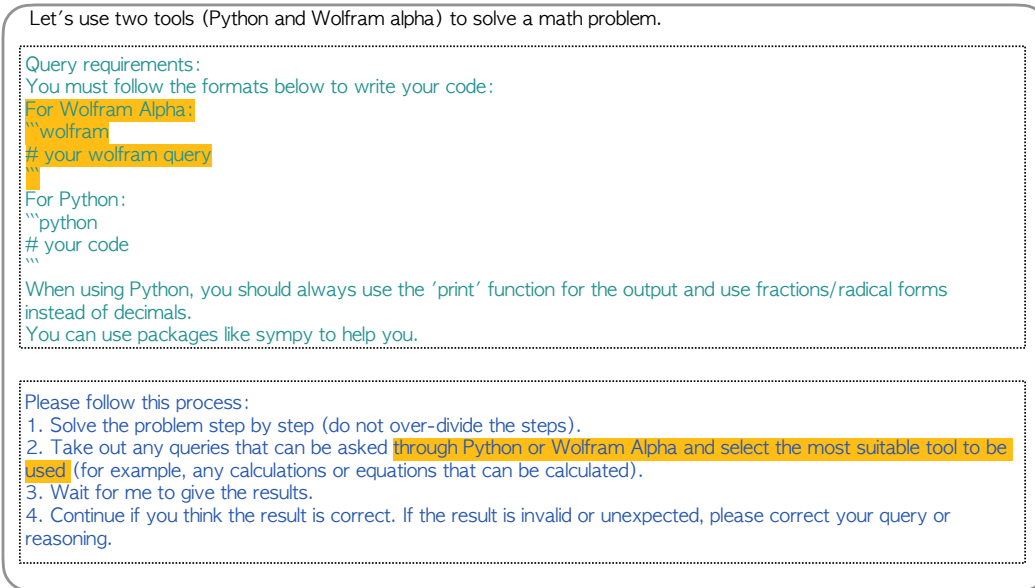


Figure 8: The Two-tools prompt used on MathChat from Section 5.2. The added requirement compared to the Python prompt is highlighted in yellow. This prompt allows the LLM agent to choose from Python or Wolfram Alpha.

only slightly increases the performance. We add a further evaluation on all the level-5 problems and find the PoT with the system message has an overall accuracy of 35.82%, which is lower than the accuracy of PoT without the system message (37.67% as shown in the main results in Table 1). We hypothesize the difference in the prompt format across different methods is the reason for this behavior. The method with the Vanilla prompt imitates a conversation between the LLM agent and humans via natural language, but PoT prompt is in Python code format, which explicitly directs the model for code completion. Thus, the system message "you are a helpful assistant" is more suitable for Vanilla prompt but doesn't align with PoT. More investigation is needed to understand the effect of system messages.

	Algebra	C.Prob	I.Alg	N.Theory	Prealg	Precalc	Total
Problem Count	50	50	50	50	50	50	300
MathChat	30	24	8	34	28	10	134
PS	17	19	12	31	26	5	110
PoT w/o sys	20	19	9	24	24	7	103
PoT w/ sys	18	23	9	23	29	7	109
Vanilla w/o sys	14	4	0	4	13	1	35
Vanilla w/ sys	26	13	1	17	21	1	79
Few-shot (k=3)	21	6	2	18	24	1	72

Table 5: Results for few-shot prompt, PoT w/ and w/o system message, Vanilla prompt w/ and w/o system message.

## D SUPPLEMENTARY FAILURE ANALYSIS

### D.1 FAILURE UNDER DIFFERENT FORMS OF PROBLEM-SOLVING PROCESSES IN MATHCHAT

The default prompt in MathChat allows the LLM agent to choose from different forms of problem-solving processes to solve the problem, and we investigate how choosing different forms could affect the performance. We plot the correct rate when a problem from each category is solved with three forms of problem-solving approaches depending on the existence and validity of the query in the

Write a python program that answers the following question: A point  $(2,n)$  is 5 units away from the point  $(-1,1)$ . What is the product of all possible integer values for  $n$ ? Problem

You can solve this problem using the distance formula between two points:  
 $distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$   
 In this case, the distance is 5 units, and the two points are  $(2, n)$  and  $(-1, 1)$ :  
 $5 = \sqrt{(2 - (-1))^2 + (n - 1)^2}$   
 Now, you can write a Python program to find all the possible integer values of  $n$ :

```

from math import sqrt

def find_possible_n_values():
    possible_n_values = []

    for n in range(-10, 10):
        distance = sqrt((2 - (-1))**2 + (n - 1)**2)
        if distance == 5:
            possible_n_values.append(n)

    return possible_n_values

def find_product_of_n_values(possible_n_values):
    product = 1
    for n in possible_n_values:
        product *= n

    return product

possible_n_values = find_possible_n_values()
product = find_product_of_n_values(possible_n_values)

print("The possible integer values of n are:", possible_n_values)
print("The product of these possible integer values is:", product)

```

This Python program first defines a function 'find\_possible\_n\_values' that iterates over a range of integer values for  $n$ , calculates the distance using the given formula, and checks whether the distance is equal to 5. It then appends the valid integer values of  $n$  to a list of possible  $n$  values. Another function, 'find\_product\_of\_n\_values', is then values of  $n$  ... [\(Explanation of Code\)](#) GPT-4

The possible integer values of  $n$  are: `(-3, 5)`  
 The product of these possible integer values is: `-15`  
 Please put the final answer in `\boxed{\}`. User Proxy Agent

The product of the possible integer values of  $n$  is: `\boxed{-15}`

Figure 9: An example of the process of PS. The query result will be returned to the LLM assistant and ask it to put the answer in box. The process of PS is exactly the same as MathChat when the agent in MathChat chooses to solve the problem with one Python program.

```

import math
import numpy as np
import sympy as sp

# Question: {problem}
# Answer this question by implementing a solver() function.
def solver():
    # Let's write a Python program step by step, and then return the answer
    # Firstly, we need define the following variable:

```

Figure 10: PoT prompt. Comparing to the original prompt from Chen et al. (2022), we add "import sympy as sp" that gives the LLM agent hint to use the sympy library. The placeholder "{problem}" will be replaced with the actual problem.

generated solution in Figure 11: 1. The LLM agent doesn't make any tool-using queries (Python) when solving the problem. 2. The agent makes one or more queries, but at least one query is invalid. 3. The agent makes all valid queries. It is shown in the plot that using Python correctly could significantly increase the correct rate, while doesn't use Python is worse than using Python but having invalid queries. The results in Figure 11 show that especially for intermediate algebra



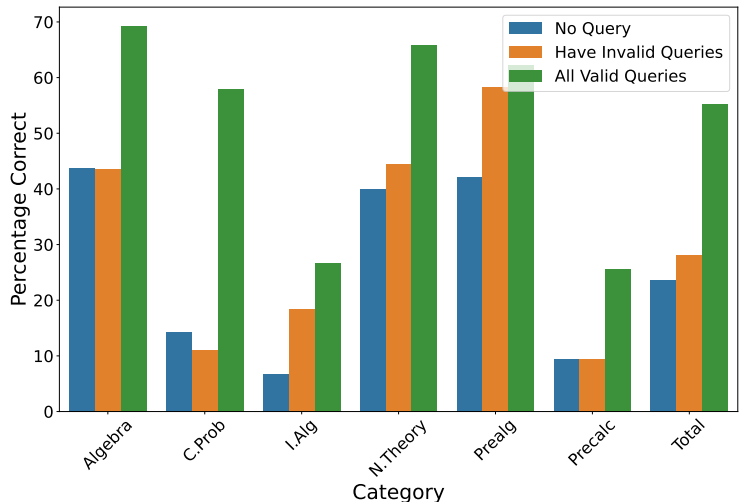


Figure 11: Success rate of MathChat under different forms of problem-solving processes: 1. the LLM agent solves the problem without making any tool-using queries. 2. The agent makes queries and has at least one invalid query 3. All queries made are valid.

**Problem:** Suppose  $x \in [-5, -3]$  and  $y \in [2,4]$ . What is the largest possible value of  $\frac{x+y}{x-y}$ ?

**LLM Agent:**

To find the largest possible value of  $\frac{x+y}{x-y}$ , we need to maximize the value of the fraction. We can first consider how the numerator and denominator are affected by the given intervals of  $x$  and  $y$ .

Since  $x \in [-5, -3]$  and  $y \in [2,4]$ , the numerator  $x + y$  will be in the range  $[-3,1]$  and the denominator  $x - y$  will be in the range  $[-9, -5]$ . Therefore, we can try to maximize the value of the fraction by choosing the largest possible value for the numerator while choosing the smallest possible value for the denominator.

Here's a Python program using the sympy package to calculate the largest possible value of the fraction  $\frac{x+y}{x-y}$ :

```

from sympy import symbols, Rational
x, y = symbols('x y')
numerator = x + y
denominator = x - y
fraction = numerator / denominator

x_range = (-5, -3)
y_range = (2, 4)

max_num = numerator.subs(x, x_range[1]).subs(y, y_range[1])
min_denom = denominator.subs(x, x_range[0]).subs(y, y_range[0])

max_value = Rational(max_num, min_denom)
print(max_value)
    
```

Please run the code and provide the result.

**User Proxy Agent:** -1/7

**Type 1 Failure:** The LLM assistant first calculates the range for numerator and denominator separately and then chooses different  $x$  and  $y$  for numerator and denominator to maximize the value, which is the wrong direction. A correct solution: Maximizing  $\frac{x+y}{x-y}$  is equivalent to maximizing  $\frac{x+y}{x-y} + 1 = \frac{2x}{x-y} = \frac{-2x}{y-x}$ . Note that  $-2x$  and  $y-x$  are always positive, so to maximize this expression, we take  $y = 2$ , the smallest possible value of  $y$ . Then maximizing  $\frac{x+2}{x-2}$  is equivalent to maximizing  $\frac{x+2}{x-2} - 1 = \frac{4}{x-2} = -\frac{4}{2-x}$ . Note that  $2-x$  is always positive, so to maximize this expression, we take  $x = -5$ . Hence, the maximum value is  $\frac{-5+2}{-5-2} = \frac{3}{7}$ .

**Problem:** The four-digit number  $25AB$  is divisible by nine, with  $A$  the tens digit and  $B$  the units digit. How many different such four-digit numbers could  $25AB$  represent?

**LLM Agent:**

Key Idea: A number is divisible by 9 if the sum of its digits is divisible by 9.

We will now write a Python program to find all the possible values of  $A$  and  $B$  such that  $25AB$  is divisible by 9. For that, we will iterate through all possible values for each digit and check if the sum of digits is divisible by 9.

```

from sympy import Symbol

# Initialize possible values
possible_values = []

# Iterate through all possible values of A (1 to 9) and B (0 to 9)
for A in range(1, 10):
    for B in range(10):
        # Calculate the sum of the digits
        sum_digits = 2 + 5 + A + B

        # Check if the sum of the digits is divisible by 9
        if sum_digits % 9 == 0:
            possible_values.append((A, B))

print(possible_values)
    
```

Now, we will run this Python code to find possible values of  $A$  and  $B$ .

**User Proxy Agent:** ((1, 1), (2, 0), (2, 9), (3, 8), (4, 7), (5, 6), (6, 5), (7, 4), (8, 3), (9, 2))

**LLM Agent:** ... Our final answer is 10.

**Type 2 Failure:** The LLM assistant has the correct idea to iterate and find all numbers that sum of digits is divisible by 9. But for  $A$ , it didn't include 0 as a possible digit, missing the number 2502 where  $(A,B) = (0, 2)$ . If change "range(1, 10)" to "range(10)", the final answer is 11, which is correct.

Figure 12: Additional example of Type 1 failure (Fail to devise a proper plan) and Type 2 failure (Fail to execute the plan flawlessly).

and prealgebra, the gap in accuracy between "using no query" and "have invalid query" is large, indicating that using Python is very helpful to solve problems from the two categories.

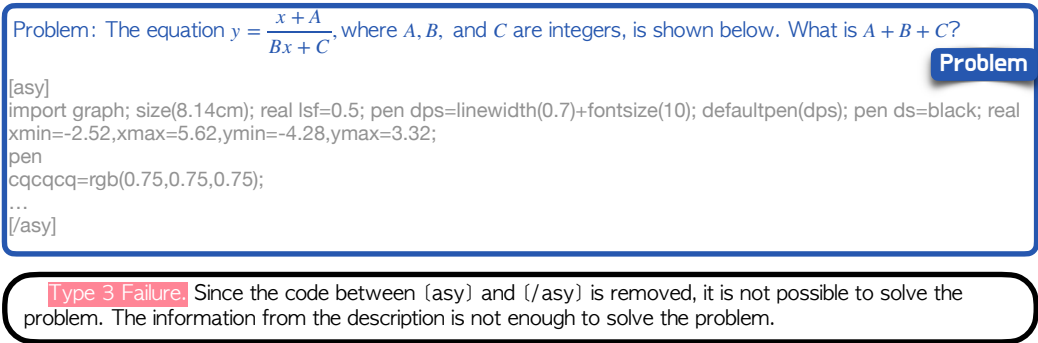


Figure 13: An example of Type 3 failure where the ASY code is removed.

### D.2 EXAMPLES OF 3 TYPES OF FAILURES

In Section 6.1, we summarize 3 main type of failures: type 1: failure to devise an appropriate plan. type 2: failure to flawlessly execute the plan. type 3: other technical errors. We give one additional example for type 1 error and type 2 error, and an example where the removal of ASY code leads to a leak of information (Figure 12, Figure 13). We note that among all the problems, the ASY code from 72 problems are removed, but 12 problems can still be solved correctly.

### D.3 FAILURE UNDER DIFFERENT METHODS

We present the precalculus example where `MathChat` fails but all other methods success (Figure 15, Figure 16, Figure 17). The results from PS and PoT show that it is easy to get this problem correct with Python (using the `sympy.simplify` function). However, in `MathChat`, the LLM agent chooses to solve the problem via direct reasoning. Both `MathChat` and vanilla prompt solve this problem by writing extremely long derivations. `MathChat` solves the problem with an even longer step-by-step response and makes a calculation error during the process.

Additionally, we also provide an overview of the number of problems where all methods either fail or succeed in Table 6.

	Algebra	C.Prob	I.Alg	N.Theory	Prealg	Precalc	Total
All Success	46	13	0	18	45	1	176
All Fail	57	32	171	20	36	86	402

Table 6: The number of problems where all methods fail, and all methods succeed.

### D.4 THE RELATIONSHIP BETWEEN FAILURE RATE AND GENERATED SOLUTION LENGTH

Chain of Thought (CoT) prompting shows that extra reasoning steps for a problem can improve the ability of LLMs Wei et al. (2022). With GPT-4, explicit reasoning is no longer an issue. Instead, we find that a long and tedious reasoning process may result in more type 2 failures, such as calculation errors, which results in a wrong answer even the overall direction is correct. We plot the distribution of correct and wrong answer lengths and also the answer length of the given solution (The length of the string list from splitting with a single space is used here). Since more complex and challenging problems are likely to have a longer solving process but still a lower success rate, we separate problems from Intermediate Algebra and Precalculus with other categories (Figure 14), to distinguish less challenging problems from harder problems. We note that the success rate of `MathChat` on the four less challenging categories goes over 50%, but the rate is lower than 20% for Intermediate Algebra and Precalculus.

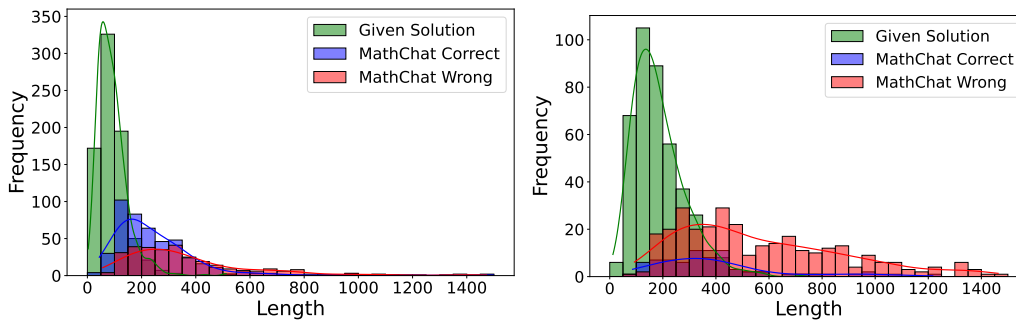


Figure 14: Distribution of solution length of both correctly and incorrectly solved problems in MathChat. The distribution of length of the given solution (ground truth) is also shown. The left figure represents the distribution of the less challenging categories and the right figure represents problems from Intermediate Algebra and Precalculus. We cut off outliers that the split string length is longer than 1500.

Overall, the solution length of MathChat is longer than the ground truth solution. The length of the given solution on the two fundamentally challenging categories is longer than other categories. For MathChat, correct answers and wrong answers from the less challenging categories have a similar distribution in solution length, where the majority of problems are solved with 50 to 500 string length. For harder problems, however, an answer with more than 600 string lengths is likely to be wrong. From the precalculus problem shown in Figure 17, the LLM agent can choose a plausible strategy to solve the problem, but that strategy is less efficient and involve more math operations compared to the given solution, this results in a much longer response, and it is more likely to make errors during the process.

Problem: Given constants  $C$  and  $D$ , suppose that  $\tan A$  and  $\tan B$  are the solutions to  $x^2 + Cx + D = 0$ , where  $\tan(A + B)$  is defined.  
Simplify  $\sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B)$ .  
Your expression should contain only one of the variables  $A$ ,  $B$ ,  $C$  and  $D$ .

Problem

By Vieta's formulas,  $\tan A + \tan B = -C$  and  $\tan A \tan B = D$ . Then from the angle addition formula,  

$$\tan(A + B) = \frac{\tan A + \tan B}{1 - \tan A \tan B} = -\frac{C}{1 - D}$$
 We write the expression we are interested in, in terms of  $\tan(A + B)$  :  

$$\begin{aligned} & \sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B) \\ &= \cos^2(A + B)\tan^2(A + B) + C \cos^2(A + B)\tan(A + B) + D \cos^2(A + B) \\ &= \cos^2(A + B)(\tan^2(A + B) + C \tan(A + B) + D) \\ &= \frac{\cos^2(A + B)}{\sin^2(A + B) + \cos^2(A + B)}(\tan^2(A + B) + C \tan(A + B) + D) \\ &= \frac{1}{\tan^2(A + B) + 1} \cdot (\tan^2(A + B) + C \tan(A + B) + D). \end{aligned}$$
 Then  

$$\begin{aligned} & \frac{1}{\tan^2(A + B) + 1} \cdot (\tan^2(A + B) + C \tan(A + B) + D) \\ &= \frac{1}{\left(-\frac{C}{1-D}\right)^2 + 1} \cdot \left(\left(-\frac{C}{1-D}\right)^2 - C \cdot \frac{C}{1-D} + D\right) \\ &= \frac{(1-D)^2}{(1-D)^2 + C^2} \cdot \frac{D(C^2 + (1-D)^2)}{(1-D)^2} \\ &= \boxed{D} \end{aligned}$$

Correct Solution

To write a Python program that answers this question, we first need to convert the mathematical expressions into Python code using math library. First, let's find the sum and product of  $\tan A$  and  $\tan B$  using the givens:

We have  $\tan A + \tan B = -C$  and  $\tan A \cdot \tan B = D$ .

Vanilla

Now, using the angle addition formula, we have:

$$\sin(A + B) = \sin A \cos B + \cos A \sin B$$

and

$$\cos(A + B) = \cos A \cos B - \sin A \sin B$$

But then, using the expressions for  $\tan$ , we can rewrite these as:

$$\sin(A + B) = \frac{\sin A}{\cos A} \cos B + \cos A \frac{\sin B}{\cos B} = \frac{\sin A \cos B \cos^2 A + \sin B \cos A \cos^2 B}{\cos^2 A \cos^2 B}$$

and

$$\cos(A + B) = \frac{\cos A}{\sin A} \sin B - \frac{\cos B}{\sin B} \sin A = \frac{\sin^2 A \sin B \cos B - \sin^2 B \sin A \cos A}{\sin^2 A \sin^2 B}$$

Now we square both expressions and add:

$$\begin{aligned} & \sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B) \\ &= \frac{((\sin A \cos B \cos^2 A)^2 + (2)(\sin A \cos A \sin B \cos B)(\sin A \cos B \cos^2 A)(\sin B \cos B \cos^2 A)) + (\sin A \cos A \sin B \cos B \cos^2 A)^2}{\cos^4 A \cos^4 B} \\ &+ C \frac{((\sin^2 A \sin B \cos B)^2 - (2)(\sin A \cos A \sin B \cos B)(\sin^2 A \sin B \cos B)(\sin^2 B \sin A \cos A) + (\sin^2 B \sin A \cos A)^2)}{\sin^2 A \sin^2 B \cos^2 A \cos^2 B} \\ &+ D \frac{((\sin^2 A \sin B \cos B)^2 + (\sin B \cos B \sin A \cos A - \sin A \cos B \sin B \cos A)(\sin A \cos A \sin B \cos B \cos^2 A) + (\sin A \cos B \sin B \cos A)^2)}{\sin^4 A \sin^4 B} \end{aligned}$$

Using the sums and products of  $\tan A$  and  $\tan B$ ,  $\tan A \cdot \tan B = D$ :

$$= \frac{\sin^2 A \cos^2 B + 2 \sin A \cos A \sin B \cos B (\tan A + \tan B) + \sin^2 B \cos^2 A - C \sin A \sin B \cos A \cos B (\tan A + \tan B) (\sin A \sin B) + D \sin^2 A \sin^2 B}{\cos^2 A \cos^2 B \sin^2 A \sin^2 B}$$

Noticing that  $\tan A + \tan B = -C$  and canceling some terms, we find that the expression simplifies to:  $\boxed{D}$

Figure 15: The precalculus problem where other methods are correct but MathChat is wrong. This figure shows the ground truth solution and the response with vanilla prompt.

Problem: Given constants  $C$  and  $D$ , suppose that  $\tan A$  and  $\tan B$  are the solutions to  $x^2 + Cx + D = 0$ , where  $\tan(A + B)$  is defined.  
Simplify  $\sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B)$ .  
Your expression should contain only one of the variables  $A$ ,  $B$ ,  $C$  and  $D$ .

To write a Python program that answers this question, we first need to convert the mathematical expressions into Python code using math libraries. Then we will simplify the given expression containing sinusoidal functions.

Here is the program:

```
import math
import cmath
import sympy as sp

# Constants
C, D = sp.symbols('C D')

# Solving the quadratic equation
x = sp.symbols('x')
quadratic_eq = x**2 + C*x + D
solutions = sp.solve(quadratic_eq, x)

# Assuming tan(A) and tan(B) are two solutions
tan_A = solutions[0]
tan_B = solutions[1]

# Using properties of tan to find A and B
A = sp.atan(tan_A)
B = sp.atan(tan_B)

# Finding sin(A+B) and cos(A+B)
sin_A_plus_B = sp.sin(A + B)
cos_A_plus_B = sp.cos(A + B)

# Simplifying the given expression
expression = sin_A_plus_B**2 + C * sin_A_plus_B * cos_A_plus_B + D * cos_A_plus_B**2
simplified_expression = sp.simplify(expression)

print(f"The given expression simplified is: {simplified_expression}")
```

PS

```
import math
import numpy as np
import sympy as sp # added

def solver():
    # Define the symbols for the constants and angles
    C, D, A, B = sp.symbols('C D A B')
    # Now, we know that tan(A) and tan(B) are solutions to the quadratic equation x^2 + Cx + D = 0
    # We can represent this equation using Sympy
    x = sp.Symbol('x')
    quadratic_eq = x**2 + C*x + D
    # Now, let's use Sympy to find the solutions
    tan_values = sp.solve(quadratic_eq, x)
    # Now, we have tan(A) and tan(B) as elements of the tan_values list
    tan_A = tan_values[0]
    tan_B = tan_values[1]
    # Next, find sin(A), cos(A), sin(B), and cos(B) from tan(A) and tan(B)
    sin_A = tan_A / sp.sqrt(1 + tan_A**2)
    cos_A = 1 / sp.sqrt(1 + tan_A**2)
    sin_B = tan_B / sp.sqrt(1 + tan_B**2)
    cos_B = 1 / sp.sqrt(1 + tan_B**2)
    # Now we have all the necessary components to find sin(A + B) and cos(A + B)
    sin_AB = sin_A * cos_B + cos_A * sin_B
    cos_AB = cos_A * cos_B - sin_A * sin_B
    # Now we can form the given expression in terms of sin(A + B) and cos(A + B)
    given_expr = sin_AB**2 + C * sin_AB * cos_AB + D * cos_AB**2
    # Simplify the given expression
    simplified_expr = sp.simplify(given_expr)
    return simplified_expr

print(solver())
```

POT

Figure 16: The precalculus problem where other methods are correct but MathChat is wrong (Continued). This figure shows the PS and PoT code. Both code returns the correct result: "D".

Problem: Given constants  $C$  and  $D$ , suppose that  $\tan A$  and  $\tan B$  are the solutions to  $x^2 + Cx + D = 0$ , where  $\tan(A + B)$  is defined.

Problem

Simplify  $\sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B)$ .

Key Idea: We will use the relationship between  $\tan A$ ,  $\tan B$ , and properties of sine and cosine.

LLM Agent

Step 1: Since  $\tan A$  and  $\tan B$  are the solutions of the given equation, we have:

$$\tan A + \tan B = -C$$

$$\tan A \cdot \tan B = D$$

Step 2: Convert the expression  $\sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B)$  to an expression in terms of  $\tan A$  and  $\tan B$ .

Using the formula for  $\sin(A + B) = \sin A \cos B + \cos A \sin B$  and  $\cos(A + B) = \cos A \cos B - \sin A \sin B$ , we have:

$$\begin{aligned} &\sin^2(A + B) + C \sin(A + B)\cos(A + B) + D \cos^2(A + B) = \\ &((\sin A \cos B + \cos A \sin B)^2 + C(\sin A \cos B + \cos A \sin B)(\cos A \cos B - \sin A \sin B) + D(\cos A \cos B - \sin A \sin B)^2) \end{aligned}$$

Continue. Please keep solving the problem until you need to query. (If you get to the answer, ...

User Proxy Agent

Step 3: We will now expand the expression and simplify it using the relations between  $\tan A$ ,  $\tan B$ , and  $C$ ,  $D$ :

$$((\sin A \cos B + \cos A \sin B)^2 + C(\sin A \cos B + \cos A \sin B)(\cos A \cos B - \sin A \sin B) + D(\cos A \cos B - \sin A \sin B)^2)$$

Expanding the terms:

$$\begin{aligned} &= (\sin^2 A \cos^2 B + 2 \sin A \cos A \sin B \cos B + \cos^2 A \sin^2 B) \\ &+ C(\sin A \cos A \cos^2 B - \sin^2 A \sin B \cos B + \cos A \cos B \cos^2 A - \cos^2 A \sin A \sin B) \\ &+ D(\cos^4 A - 2 \sin A \cos A \sin B \cos B + \sin^2 A \sin^2 B) \end{aligned}$$

Now we use the identity  $\sin^2 x + \cos^2 x = 1$

Step 4: Rewrite the expression in terms of  $\tan A$  and  $\tan B$ :

$$\begin{aligned} &= (\sin A \cos B)(\sin A \cos B + 2 \cos A \sin B) \\ &+ C(\sin A \cos^2 B \cos A - \sin^2 A \sin B \cos B + \cos A \cos B \cos^2 A - \sin B \cos^2 A \cos A) \\ &+ D(\cos A \sin B)(\cos^2 A - \sin A \cos A \cos B - \sin B) \end{aligned}$$

Now we substitute  $\sin A$  with

$$\frac{\tan A}{\sqrt{1 + \tan^2 A}} \text{ and } \cos A \text{ with } \frac{1}{\sqrt{1 + \tan^2 A}}, \text{ and similarly for } B:$$

$$= \frac{\tan A \tan B}{(\sqrt{1 + \tan^2 A} \sqrt{1 + \tan^2 B})(\tan^2 A + \tan^2 B + 2)} + C \left( \frac{\tan A + \tan B}{(\sqrt{1 + \tan^2 A} \sqrt{1 + \tan^2 B})(\tan^2 A + \tan^2 B + 1)} \right) + D \frac{\tan^2 A \tan^2 B}{(1 + \tan^2 A)(1 + \tan^2 B)}$$

We will use the relations  $\tan A + \tan B = -C$  and  $\tan A \cdot \tan B = D$ .

Step 5: Make substitutions and simplify:

$$= \frac{-D}{(\sqrt{1 + \tan^2 A} \sqrt{1 + \tan^2 B})(\tan^2 A + \tan^2 B + 2)} + C \left( \frac{-C}{(\sqrt{1 + \tan^2 A} \sqrt{1 + \tan^2 B})(\tan^2 A + \tan^2 B + 1)} \right) + \frac{D^2}{(1 + \tan^2 A)(1 + \tan^2 B)}$$

Notice that the three terms have common denominators. If we combine them, we get:

$$= \frac{-D(\tan^2 A + \tan^2 B + 1) - C^2(\tan^2 A + \tan^2 B + 2) + D^2(1 + \tan^2 A)(1 + \tan^2 B)}{(\sqrt{1 + \tan^2 A} \sqrt{1 + \tan^2 B})(\tan^2 A + \tan^2 B + 1)(1 + \tan^2 A)(1 + \tan^2 B)}$$

Step 6: Simplify the expression and obtain the final answer:

$$= \frac{-C^2}{D}$$

LLM Agent

Figure 17: The precalculus example where all the other methods are correct but MathChat is wrong (Continued). This figure shows the conversation generated in MathChat. The LLM agent in MathChat chooses to solve the problem via direct reasoning, and it makes a calculation error when expanding the terms in Step 3.