SciAgent: Tool-augmented Language Models for Scientific Reasoning

Yubo Ma[1*], Zhibin Gou[2*], Junheng Hao[3], Ruochen Xu[3], Shuohang Wang[3],
Liangming Pan[4], Yujiu Yang[2], Yixin Cao[5†] , Aixin Sun[1†]
[1] Nanyang Technological University [2] Tsinghua University [3] Microsoft
[4] University of Arizona [5] Fudan University
yubo001@e.ntu.edu.sg

Abstract

Scientific reasoning poses an excessive challenge for even the most advanced Large Language Models (LLMs). To make this task more practical and solvable for LLMs, we introduce a new task setting named *tool-augmented scientific reasoning*. This setting supplements LLMs with scalable toolsets, and shifts the focus from pursuing an omniscient problem solver to a proficient tool-user. To facilitate the research of such setting, we construct a tool-augmented training corpus named MathFunc which encompasses over 30,000 samples and roughly 6,000 tools. Building on MathFunc, we develop SciAgent to retrieve, understand and, if necessary, use tools for scientific problem solving. Additionally, we craft a benchmark, SciToolBench, spanning five scientific domains to evaluate LLMs' abilities with tool assistance. Extensive experiments on SciToolBench confirm the effectiveness of SciAgent. Notably, SciAgent-Llama3-8B surpasses other LLMs with the comparable size by more than 8.0% in absolute accuracy. Furthermore, SciAgent-DeepMath-7B shows much superior performance than ChatGPT.

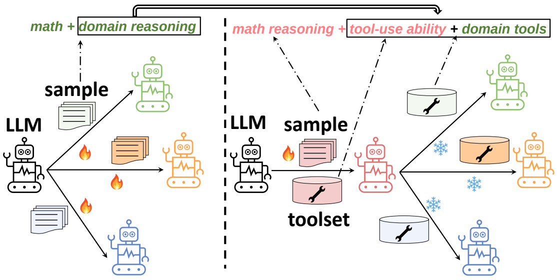


Figure 1: Two paradigms for scientific reasoning task. Different colors represent different scientific domains. **Left:** Supervised fine-tuning setting. Collecting annotations and fine-tuning LLMs domain by domain. **Right:** Our proposed *tool-augmented* setting. We decompose the scientific reasoning ability into three aspects: math reasoning ability, tool-use ability and domain-specific knowledge. LLMs are fine-tuned on math-related, tool-augmented samples (color in red). When adapting LLMs to a specific domain, a pluggable and domain-specific toolset is attached to provide the domain knowledge. No additional fine-tuning is further required.

1 Introduction

Scientific reasoning (Ouyang et al., 2023; Zhao et al., 2023) aims to comprehend and make decisions regarding problems among STEM (*Science, Technology, Engineering and Mathematics*) domains. It is a fundamental aspect of intelligence, a demanding capability of Large Language Models (LLMs), and a notoriously challenging task. For instance, even GPT-4 (OpenAI, 2023) achieves only 50% and 35% accuracy on TheoremQA (Chen et al., 2023b) and SciBench (Wang et al., 2023b), respectively. Regarding open-source LLMs such as Llama3 (LlamaTeam, 2024) and Mistral (Jiang et al., 2023), the performances are only about 30% accuracy or even less.

The challenge in scientific reasoning arises from the need for both mathematical (math) and domain-specific reasoning abilities. To address the physical problem in Figure 3, for example, it is necessary to both understand *Malus' law* (domain knowledge) for analyzing the intensity of polarized light, and possess quantitative ability for calculating the light intensity ratios. A natural approach involves collecting annotations and fine-tuning LLMs to enhance their math and domain-specific reasoning abilities, as depicted in Figure 1 (left). However, annotating scientific reasoning problems is extremely expensive. What is worse, adapting LLMs to a new domain demands a fresh round of annotation and fine-tuning, rendering this approach impractical.

In this paper, we draw inspirations from *tool learning* (Qin et al., 2023a) to enhance LLMs' scientific reasoning capabilities. Instead of solving

[*] This work is done during Yubo and Zhibin's internship at Microsoft.

[†] Co-corresponding authors.

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 15701–15736
November 12-16, 2024 ©2024 Association for Computational Linguistics

scientific problem from scratch, humans have summarized and wrapped various points as generalized and well-documented functions in scientific computing softwares, such as Matlab[1], WolframAlpha[2], SymPy[3], *etc*. These functions[4], which could be equivalently viewed as external tools, greatly facilitate math-adept users to solve difficult scientific problems. In analogy with humans, we do not pursue an omniscient **solver** across various scientific domains. Instead, we assume the access to domain-specific toolsets and purse a unified, generalized LLM-based **tool-user** as shown in the Figure 1 (right). This approach tackles domain-specific reasoning challenges by enabling LLMs learn to use a reusable and scalable toolkit. It alleviates the reasoning challenges of LLMs by concentrating solely on enhancing their tool-use abilities. These abilities are not only easier to acquire but also applicable across a variety of scientific fields. By attaching domain-specific toolsets, our tool-users can be readily adapted to different fields without the need for additional in-domain fine-tuning.

This work focuses on developing and benchmarking the ability of LLMs in scientific reasoning **with the help of tools**. We envision a scenario where LLMs have access to a domain-specific toolset, comprising various specialized functions. Upon this scenario, we propose a complete framework of dataset construction, model training and evaluation. Given a scientific question, LLMs are supposed to retrieve functions from the toolset and optionally incorporate functions into the formulated solution. We employ an automatic pipeline featuring GPT-4/-4o to compile a large-scale, math-related, tool-augmented training corpus named as MATHFUNC. This corpus is designed to enable LLMs to learn both essential math skills and how to retrieve, understand and use functions properly. As a result, MATHFUNC contains 31,375 samples and equipped with a toolset encompassing 5,981 generalized and well-documented functions. We detail this training corpus in Section 3.

We fine-tune open-source LLMs on MATHFUNC to develop tool-augmented agents named SCIAGENT detailed in Section 4. As shown in Figure 3, SCIAGENT firstly generate a *high-level planning* in response to a given question. The agents then use this plan, along with the question, to retrieve functions from the given toolset. Leveraging these retrieved functions, the agents further complete the *low-level action* integrating natural language and Python code. Finally the agents execute the code to complete the problem at hand.

To benchmark the tool-use abilities in scientific reasoning, we develop a new benchmark named SCITOOLBENCH as described in Section 5. Building upon TheoremQA (Chen et al., 2023b) and SciBench (Wang et al., 2023b), it has 4,250 questions covering five domains: *Mathematics*, *Physical*, *Chemistry*, *EECS*, and *Finance*. It also contains five domain-specific toolsets comprising a total of 2,285 functions. We evaluate SCIAGENT on SCITOOLBENCH and another benchmark derived from CREATOR-challenge (Qian et al., 2023). Experimental results demonstrate that our agents present remarkable scientific reasoning capabilities. Notably, SCIAGENT-LLAMA3-8B surpasses the best comparable open-source LLMs by an absolute 8.0% accuracy, and SCIAGENT-DEEPMATH-7B outperforms ChatGPT by a large margin. We also conduct an extensive analysis of the benefits and limitations of SCIAGENT series, providing valuable insights for future research.

2 Preliminary

Related Work. Current methods (Chen et al., 2023b; Xu et al., 2023b; Ouyang et al., 2023), especially those based on open-source LLMs, perform far from satisfactory on scientific reasoning benchmarks (Chen et al., 2023b; Wang et al., 2023b). We attribute it to the scarcity of annotated samples across diverse scientific domains. As a comparison, LLMs present much more remarkable performance on math problems (Yue et al., 2023b; Gou et al., 2023b; Azerbayev et al., 2023) due to the abundant training corpora and/or annotations. Different from concurrent work (Zhang et al., 2024) which collects physics and chemistry annotations, we do not pursue a problem-solver on some specific scientific domains. Instead, we consider to develop a generalized tool-user being proficient on solving diverse scientific problems with the aid of tools. Following previous work on math domain (Qian et al., 2023; Cai et al., 2023; Yuan et al., 2023a), the tools here refer to Python functions. Please see more detailed literature review in Appendix A.

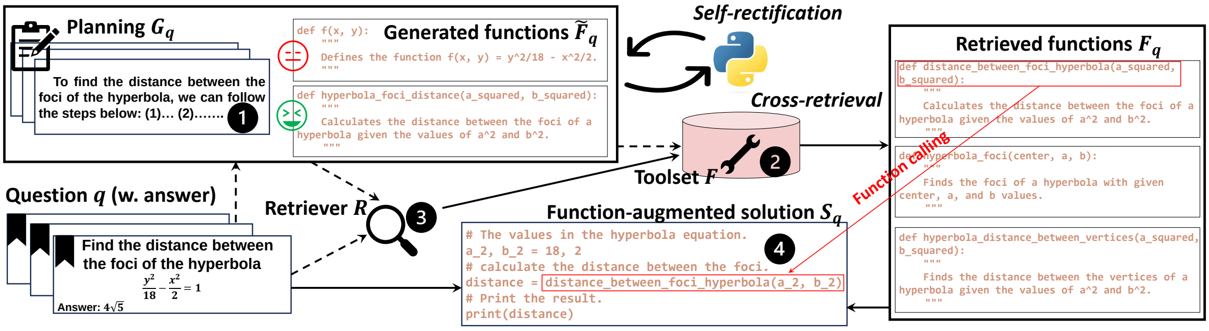
Task Formulation. Given a scientific domain D (*e.g.,* physics), *tool-augmented* scientific reasoning

[1] https://www.mathworks.com/products/matlab.html

[2] https://www.wolframalpha.com/

[3] https://www.sympy.org/en/index.html

[4] In this work, tools refer to Python functions. We use tools and functions interchangeably unless otherwise specified.


Planning G_q
To find the distance between the foci of the hyperbola, we can follow the steps below: (1)… (2)…….
1
Generated functions $\widetilde{F}_q$
Self-rectification
Cross-retrieval
Toolset F
2
Retrieved functions F_q
Question q (w. answer)
Find the distance between the foci of the hyperbola
$\frac{y^2}{18} - \frac{x^2}{2} = 1$
Answer: $4\sqrt{5}$
Retriever R
3
Function-augmented solution S_q
4
Function calling

```
def f(x, y):
    """
    Defines the function f(x, y) = y^2/18 - x^2/2.
    """
```

```
def hyperbola_foci_distance(a_squared, b_squared):
    """
    Calculates the distance between the foci of a
hyperbola given the values of a^2 and b^2.
    """
```

```
# The values in the hyperbola equation.
a_2, b_2 = 18, 2
# calculate the distance between the foci.
distance = distance_between_foci_hyperbola(a_2, b_2)
# Print the result.
print(distance)
```

```
def distance_between_foci_hyperbola(a_squared,
b_squared):
    """
    Calculates the distance between the foci of a
hyperbola given the values of a^2 and b^2.
    """
```

```
def hyperbola_foci(center, a, b):
    """
    Finds the foci of a hyperbola with given
center, a, and b values.
    """
```

```
def hyperbola_distance_between_vertices(a_squared,
b_squared):
    """
    Finds the distance between the vertices of a
hyperbola given the values of a^2 and b^2.
    """
```

Figure 2: Automatic pipeline for MATHFUNC construction. Please view it starting from the bottom left corner and proceed clockwise. We disentangle the constructions of toolset (dashed lines) and function-augmented samples (solid lines) for more generalized annotations. We do not visualize the function-free samples for simplicity.

task assumes access to (1) a question $q \in D$ and (2) a toolset F_D. F_D encompasses large amounts of well-documented, domain-specific functions $\{f_1, ..., f_m\}$. Our objective is to develop an agent $\mathcal{M}$ which selectively use functions in F_D to enhance the answering for the question q.

3 Training Corpus: MATHFUNC

To our best knowledge, there are no readily available tool-augmented datasets in scientific reasoning domains. Therefore, we construct a corpus named MATHFUNC teaching LLMs to better understand and use functions. MATHFUNC is composed of (1) a toolset F[5] including 5,981 generalized, well-documented, math-related functions and (2) a dataset D encompassing 31,375 samples in which solutions call the function from the toolset if necessary (*e.g.,* ④ in Figure 2). We build this corpus based on MATH (Hendrycks et al., 2021b) training set because we expect to teach LLMs both math skills and tool-use abilities.

Sample Format. Each sample is a quintuple (q, G_q, F_q, S_q, a_q). Here q is a question, G_q is the planning, F_q is the function set filtered from the toolset ($F_q \subset F$, $|F_q| \ll |F|$), S_q is the solution and a_q is the answer. S_q interleaves rationales E_q[6] and programs P_q which optionally call functions in F_q to facilitate the problem solving.

We employ an automatic pipeline to construct MATHFUNC. We illustrate the pipeline in Figure 2 and detail the process in the following subsections.

3.1 Planning and Toolset Construction

This module is depicted in the top-left side of Figure 2. Given a question q and its ground-truth solution (written in pure natural language) in MATH training set, we ask GPT-4 to generate (1) a high-level planning G_q to analyze this question, (2) one or more well-documented functions $\tilde{F}_q$ and (3) a solution $\tilde{S}_q$ calling the functions above. The prompt used is shown in Appendix H.1. In the prompt, we emphasize that the functions should be as **composable and generalized** as possible. Specifically, we do not hope that each question generates only one ad-hoc function (which could only be used by this question). Instead, we expect GPT-4 to generate functions that follow the points in the planning G_q and can be reused by other questions. Following previous work (Qian et al., 2023; Pan et al., 2023), we provide the error feedback to GPT-4 if the solutions fail to execute, and ask GPT-4 to rectify the errors in $\tilde{F}_q$ or $\tilde{S}_q$. We repeat this procedure until successful execution or reaching maximum loop limitation. The prompt used for rectification is shown in Appendix H.2.

We collect G_q (① in Figure 2, the same below) and add $\tilde{F}_q$ to the toolset (②) for question q if the rectified solution $\tilde{S}_q$ leads to the correct answer $\tilde{a}_q$. Regarding the toolset, it is iterated on all questions and finally accumulated as below:

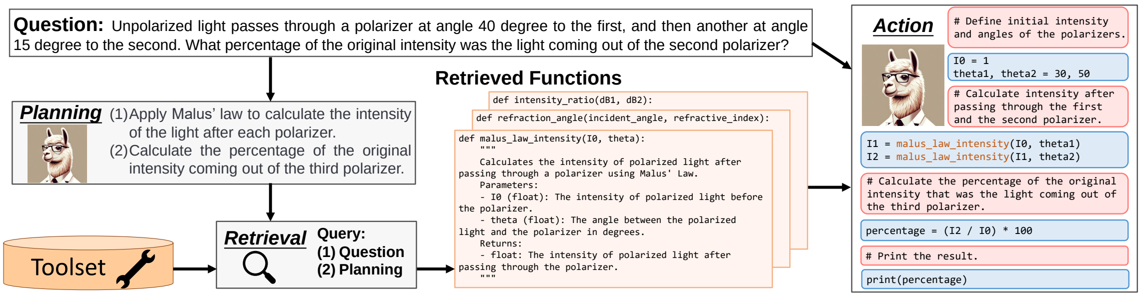
$$F = \bigcup_{q \in D} \tilde{F}_q \cdot \mathrm{I}(\tilde{a}_q \text{ is correct})$$

3.2 Function-augmented Solutions

To collect function-augmented solution S_q and F_q, a natural idea is to directly use the $\tilde{S}_q$ and $\tilde{F}_q$ generated above. However, we find that $\tilde{S}_q$ tends to

[5] We remove the domain-specific subscript D for expression simplicity. The same below.

[6] Here E_q is written in natural language but formatted as the annotation lines in the program.



Question: Unpolarized light passes through a polarizer at angle 40 degree to the first, and then another at angle 15 degree to the second. What percentage of the original intensity was the light coming out of the second polarizer?

Planning (1) Apply Malus' law to calculate the intensity of the light after each polarizer. (2) Calculate the percentage of the original intensity coming out of the third polarizer.

```
def intensity_ratio(dB1, dB2):
```

```
def refraction_angle(incident_angle, refractive_index):
```

```
def malus_law_intensity(I0, theta):
    """
    Calculates the intensity of polarized light after
passing through a polarizer using Malus' Law.
    Parameters:
    - I0 (float): The intensity of polarized light before
the polarizer.
    - theta (float): The angle between the polarized
light and the polarizer in degrees.
    Returns:
    - float: The intensity of polarized light after
passing through the polarizer.
    """
```

```
# Define initial intensity
and angles of the polarizers.
I0 = 1
theta1, theta2 = 30, 50
# Calculate intensity after
passing through the first
and the second polarizer.
I1 = malus_law_intensity(I0, theta1)
I2 = malus_law_intensity(I1, theta2)
# Calculate the percentage of the original
intensity that was the light coming out of
the third polarizer.
percentage = (I2 / I0) * 100
# Print the result.
print(percentage)
```

Figure 3: The model architecture of SciAgent. Given a domain-specific toolset, our agent answers the question through four consecutive modules. (1) **Planning**: provides a high-level plan for this problem. (2) **Retrieval**: retrieves related functions from attached toolset. (3) **Action**: generates a low-level solution interleaving rationale and program. The program uses the retrieved functions if necessary. (4) **Execution**: calls Python executor to run the program and outputs the final answer. Not included in this figure for simplicity.

be contrived and specifically tailored to fit the requirements of function-calling. Moreover, some functions in $\tilde{F}_q$ tend to be ad-hoc[7]. For examples, the function `f(x, y)` in Figure 2 merely parameterizes the hyperbola for a specific question. Therefore we disentangle the construction of toolset and function-augmented solutions. Given the developed toolset, we design a **cross-retrieval** strategy to retrieve more generalized functions F_q and generate more qualified solutions S_q. Specifically, we remove $\tilde{F}_q$ from F temporarily and then retrieve new functions $F_q \subseteq (F \backslash \tilde{F}_q)$ for question q. This strategy eliminates the likelihood of calling ad-hoc functions from $\tilde{F}_q$ in S_q. See examples of retrieved functions, all of which are derived from other questions, in the right side of Figure 2.

Retriever. The cross-retrieval strategy necessities a retriever because it is impractical to enumerate thousands of functions in $F \backslash \tilde{F}_q$. We train a dense retriever R (③ in Figure 2). We concatenate the question q and the generated planning G_q as the query, and view the generated functions $\tilde{F}_q$ in Section 3.1 as the keys. See details about R in Appendix B.1.

Solution Generation. Upon the toolset F and the retriever R, we retrieve three functions as F_q:

$$F_q = R([q, G_q]; F \backslash \tilde{F}_q)$$

Then we employ GPT-4 to write solutions which optionally call functions in F_q to generate the solution S_q (④). The prompt used is illustrated in Appendix H.3. We explicitly point out in the prompt that $f \in F_q$ should be called if and only if when they do lower the difficulty of problem solving. It mitigates the over-exploitation of function calling in S_q and increases the robustness of models fine-tuned on these samples. Specifically, we firstly use GPT-4 with greedy decoding to generate solutions. For those failing to yield correct answers, we further apply nucleus sampling (Holtzman et al., 2020) with 5 repeat times and 0.6 temperature. We filter wrong solutions and collect remaining 6,229 samples as our function-augmented solutions.

In parallel, we use GPT-4 to generate function-free solutions. Though not indispensable, we expect them to further enhance the math reasoning, and accordingly the scientific reasoning, abilities of LLMs. We collect a total of 24,946 function-free solutions nucleus sampling with 5 repeat times and 0.6 temperature. These samples share similar format as ToRA-corpus (Gou et al., 2023b), and do not retrieve/use any functions, *i.e.*, $F_q = \varnothing$.

4 Model: SciAgent

We develop SciAgent for tool-augmented scientific reasoning task. It could make plan, retrieve functions, and leverage retrieved functions to facilitate the reasoning. We describe its inference procedure and training approach as below.

4.1 Overview

As shown in Figure 3, SciAgent comprises four successive modules.

Planning. This module provides a high-level profile for each question: $G_q = \mathcal{M}_{\text{planning}}(q)$. Such planning instructs a more targeted retrieval process.

Retrieval. Given the question and generated planning G_q, the retriever $\mathcal{M}_{\text{retrieval}}$ is introduced to retrieve related functions from the domain-specific toolset: $F_q = \mathcal{M}_{\text{retrieval}}([q, G_q]; F_D) \subseteq F_D$.

[7] Despite we instruct GPT-4 to avoid generating ad-hoc functions, there are still some ad-hoc functions in $\tilde{F}_q$

Action. This module aims to generate low-level solutions. Specifically, the agent produces $S_q = \mathcal{M}_{\text{action}}(q; F_q)$. The solution S_q is interleaved with natural language rationale E_q and program snippet P_q. The program P_q call retrieved functions with proper arguments if necessary.

Execution. This module is simply a Python Executor to run the program P_q for the final answer: $a_q = \text{Python-Executor}(P_q)$.

4.2 Training

Language models are used in three out of four modules in SciAgent: planning, retrieval and action. Rearding retrieval, we directly use the retriever R fine-tuned in Section 3.2 as $\mathcal{M}_{\text{retrieval}}$. For planning and action modules, they share the same LLMs: $\mathcal{M} = \mathcal{M}_{\text{planning}} = \mathcal{M}_{\text{action}}$. We fine-tune $\mathcal{M}$ with different instructions to make it act as planning and action modules, respectively. We construct instructions from $d = (q, G_q, F_q, S_q, a_q)$ in MathFunc.

$$D_{\text{planning}} = \{(I_{\text{plan}}(q), G_q) | d \in D\}$$
$$D_{\text{action}} = \{(I_{\text{action}}(q, F_q), S_q) | d \in D\}$$

Here I_{plan} and I_{action} are instruction templates for planning and action modules. We show these instructions in Appendix B.2, and mix up them as the training set $D = (D_{\text{planning}} \bigcup D_{\text{action}})$. Then we apply imitation learning on D to fine-tune $\mathcal{M}$.

$$L_{\mathcal{M}} = \sum_{(X,Y) \in D} -\text{log}\mathcal{P}(Y|X)$$

Implementation We detail the training process of (1) the retriever $\mathcal{M}_{\text{retrieval}}$ and (2) the planner and actor $\mathcal{M}$ in Appendix B.1 and B.2, respectively.

5 Benchmark: SciToolBench

There currently exists no benchmark assessing the scientific reasoning capabilities of LLMs **when aided by tools**. To address this gap, we develop a benchmark called SciToolBench. Our benchmark covers five domains: *Mathematics (math)*[8], *Physics*, *Chemistry*, *Finance*, *Electrical Engineering and Computer Science (EECS)*. Each domain is composed of a set of questions and a domain-specific toolset. The toolset consists of abundant generalized, high-quality and well-documented functions. We expect LLMs to retrieve, understand and, if necessary, use functions in it for reasoning.

5.1 Dataset Overview.

The statistics of SciToolBench are presented in Table 1. We leave more detailed statistics in Appendix E.2. Briefly, our benchmark comprises a total of 4,250 questions and 2,285 functions spanning across 5 scientific domains. SciToolBench differs from previous tool-based benchmarks, such as Creation Challenge (Qian et al., 2023), in several aspects: (1) Our benchmark encompasses a diverse range of scientific domains. (2) The provided tools are both composable and generalized across different questions. As indicated in Table 1, each question requires an average of 1.51 functions for resolution. And over 500 functions are designed to be applicable to two or more questions, such as `integrate_function` in math domain, `coulombs_law` in physical domain, and `calculate_pressure_van_der_waals` in chemistry domain, *etc*. It signifies that the functions in our toolset are not ad-hoc solutions tailored for specific questions. Instead, the effective utilization of the toolset demands significant reasoning abilities of tool-augmented LLMs. Thus we claim this benchmark challenging and practical.

Table 1: The statistics of our benchmark. **#H.A./#Syn.**: The number of human-annotated/synthesized questions. **#Pos./ #Neg.**: The number of positive/negative functions in the toolset. **FPQ** (function per question): The number of derived positive functions from each question. Counted on H.A. questions only.

	Question		**Function**		
	# Question	#H.A. / #Syn.	# Function	#Pos. / #Neg.	Avg. FPQ
Math	2031	434 / 1597	964	403 / 561	1.47
Physics	855	156 / 699	516	225 / 291	1.63
Chemistry	639	118 / 521	349	138 / 211	1.34
Finance	369	66 / 303	245	89 / 156	1.62
EECS	356	82 / 274	211	87 / 124	1.68
All	4250	856 / 3394	2285	942 / 1343	1.51

5.2 Dataset Annotation

We design a semi-automatic pipeline shown in Figure 4 to annotate the benchmark. It employs both GPT-4 and human annotators to combine their merits. This pipeline includes two subsequent modules: question collection and toolset construction. We introduce these modules as below and leave their details in Appendix D.

Question Collection: Our benchmark comprises 4,250 questions from two sources. (1) *Human-annotated:* We curate 856 questions from Theo-

[8] Our benchmark contains college-level questions on calculus, differential equations, group theory, *etc*, which are different from the questions in our training corpus MathFunc.

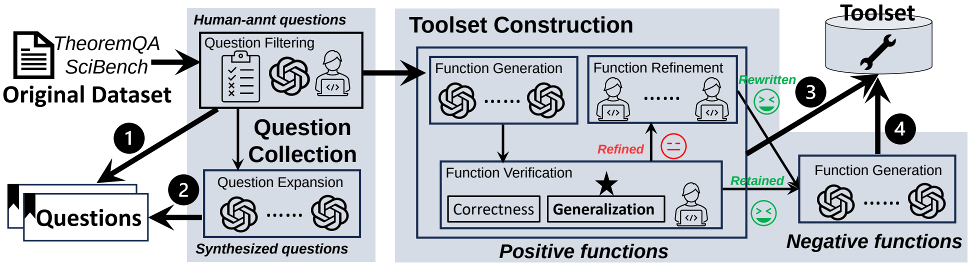


Figure 4: Semi-automatic annotation pipeline for SciToolBench. : GPT-4. : Human annotator.

remQA (Chen et al., 2023b) and SciBench (Wang et al., 2023b) (① in Figure 4, the same below). (2) *Synthesized:* To further expand the question set, we use these 856 questions as seed examples and automatically generate another 3,394 synthesized questions (②).

Toolset Construction: We construct domain-specific toolsets via two cascade modules: positive and negative function construction. (1) *Positive functions*: We define positive functions (③) as functions directly deriving from questions. The 1,216 candidate positive functions are firstly generated from GPT-4. Then human annotators carefully check them and rewrite and/or remove the unqualified ones. Both the correctness and generalization of these auto-generated functions are evaluated in this human checking process. We finalize a total of 942 positive questions. (2) *Negative functions*: We further automatically construct 1,343 negative functions (④) based on positive functions. These negative functions are apparently similar as positive ones but can not directly solve questions in our benchmark. We add them to reduce the shortcuts about question-function matching as much as possible. We finally combine both positive and negative functions as the toolset, including a total of 2,285 functions, in our benchmark.

6 Experiments

6.1 Setup

We conduct experiments on SciToolBench to evaluate the tool-augmented scientific reasoning abilities of LLMs. We report results categorized by both question domains and construction methods for fine-grained analysis. We also employ CREATION Challenge (Qian et al., 2023) as the second benchmark. It comprises 2,047 samples, with each sample consisting of a question and a ground-truth function. We re-purpose all functions to assemble a global toolset (thus including 2,047 functions). We report accuracy as the metric in all experiments.

6.2 Baselines

We compare SciAgent series with six open-source LLMs: (1) CodeLlama (Rozière et al., 2023), (2) MAmmoTH-Coder (Yue et al., 2023b), (3) ToRA-Coder (Gou et al., 2023b), (4) Mistral (Jiang et al., 2023), (5) Deepseek-Math (Shao et al., 2024), (6) Llama-3 (Touvron et al., 2023). We also list the performance of ChatGPT and GPT-4 for reference. For fair comparison, we provide all LLMs the same retriever in Section 3.2 to retrieve functions from toolset (if attached). Please see more details in Appendix C.

6.3 Main Results

We fine-tune CodeLlama, Mistral, Llama-3 and Deepseek-Math for different SciAgent variants. We present their results, along with associated baselines, in Table 2 and draw following conclusions:

The importance of math skills. The LLMs pretrained on math-related corpus, *i.e.,* Deepseek-Math series, present more competitive performance than others. And the models fine-tuned on math-related datasets from CodeLlama, *i.e.,* ToRA- and MAmmoTH-Coder, perform better than CodeLlama itself by 5.5% absolute accuracy. It presents the importance of essential math skills among diverse scientific domains.

The necessity of tool-augmented learning. Most evaluated LLMs are not inherently proficient at using tools. When equipped with toolsets, the performance of LLMs that have not undergone tool-augmented learning degrades significantly. For instance, the 7∼8B models such as ToRA-Coder, Mistral, Deepseek-Math, and Llama-3 show performance drops of 3.0%, 0.9%, 4.9%, and 3.7%, respectively. As shown in Figure 5, LLMs demonstrate proficient tool-use abilities and benefit from the attached toolsets only when they have undergone tool-augmented learning, *i.e.,* fine-tuning on MathFunc. As a result, our agents outperform other open-source LLMs by a large margin. No-

Table 2: Main results on two benchmarks. We highlight our SciAgent series in purple. The best results (among all open-source LLMs, the same below) are in bold face and the second best are underlined.

Model	Size	Toolset	CREATION	SciToolBench										
				HUMAN-ANNOTATED					SYNTHESIZED					ALL
				Math	Physics	Chemistry	Finance	EECS	Math	Physics	Chemistry	Finance	EECS	
ChatGPT	-	✗	54.6	33.4	19.2	18.6	53.0	25.6	36.3	22.6	23.0	35.3	36.5	31.0
		✓	59.8	32.0	31.4	33.9	53.0	48.8	32.6	27.5	22.5	25.9	35.9	30.9
GPT-4	-	✗	60.0	52.8	42.9	47.5	65.2	35.4	62.3	58.4	54.3	69.6	66.7	59.2
		✓	69.8	63.1	63.5	63.6	80.3	80.5	60.0	59.1	50.2	58.2	58.7	59.7
CodeLlama	7B	✗	17.7	6.5	0.6	5.1	4.9	7.6	10.4	2.7	3.6	8.8	6.9	6.9
CodeLlama	7B	✓	26.1	9.2	8.3	10.2	24.2	25.6	6.6	4.3	4.2	8.0	12.2	7.4
ToRA-Coder	7B	✗	29.7	26.3	4.5	6.8	9.1	24.4	18.8	7.0	7.1	10.5	10.6	14.2
ToRA-Coder	7B	✓	21.4	21.7	4.5	5.1	13.6	15.9	17.8	9.6	9.9	9.9	13.9	11.2
MAmmoTH-Coder	7B	✓	21.6	14.8	18.5	11.0	25.8	40.0	14.3	7.3	6.7	13.1	12.9	12.8
Mistral	7B	✗	30.1	11.3	9.6	7.6	18.2	13.4	19.2	10.7	9.4	16.8	18.5	14.8
Mistral	7B	✓	27.6	13.1	13.5	14.4	34.8	19.5	10.4	15.2	13.1	22.6	15.2	13.7
Deepseek-Math	7B	✗	44.7	26.5	19.2	17.8	27.3	20.7	31.6	21.9	23.6	28.8	24.8	26.7
Deepseek-Math	7B	✓	41.3	24.2	24.4	25.4	43.9	42.7	19.8	21.6	17.7	24.1	20.8	21.8
Llama-3	8B	✗	40.3	28.1	10.9	16.9	27.3	25.6	32.7	18.3	21.3	27.4	24.1	26.0
Llama-3	8B	✓	38.0	24.7	26.9	25.4	42.4	37.8	20.2	19.7	18.4	24.8	28.4	22.3
SciAgent-Coder	7B	✓	53.0	30.0	28.3	24.6	39.3	**57.3**	29.8	20.1	22.9	26.3	29.7	27.6
SciAgent-Mistral	7B	✓	54.0	31.3	33.3	33.9	48.5	51.2	30.3	25.6	21.9	35.8	36.6	30.3
SciAgent-Llama3	8B	✓	58.2	34.3	41.0	35.6	56.1	56.1	34.9	32.2	29.4	35.4	34.6	34.7
SciAgent-DeepMath	7B	✓	**60.4**	**41.2**	**54.5**	**44.9**	**57.5**	51.2	**37.1**	**40.1**	**36.5**	**43.1**	**40.2**	**40.0**
CodeLlama	13B	✗	23.0	9.9	3.2	1.7	9.1	6.1	13.5	4.4	4.8	8.8	12.9	9.3
CodeLlama	13B	✓	38.9	12.7	14.7	7.6	33.3	34.1	9.0	6.4	4.4	12.4	11.9	9.8
ToRA-Coder	13B	✗	30.9	28.6	3.8	4.2	16.7	30.5	22.6	9.0	8.5	13.1	16.5	17.1
ToRA-Coder	13B	✓	28.0	32.0	2.6	11.9	24.2	35.4	17.9	12.9	11.7	13.9	14.2	16.9
MAmmoTH-Coder	13B	✓	34.7	21.4	18.6	11.0	25.8	39.0	20.4	12.7	10.7	15.3	25.1	18.2
SciAgent-Coder	13B	✓	54.4	35.0	32.1	28.8	42.4	51.2	30.9	25.0	22.6	30.6	30.0	29.8

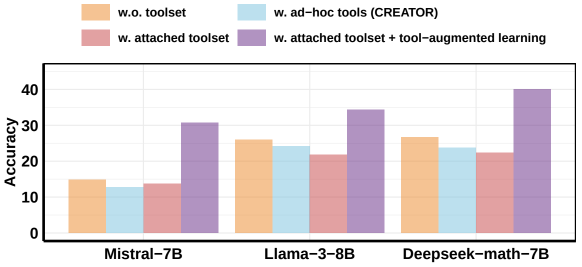


Figure 5: Evaluated LLMs are not native tool-users. Their performance drops when they are equipped with either self-derived or external toolsets (color in blue and red, respectively). Tool-augmented learning (color in purple; fine-tuning on MathFunc) makes them benefit from attached toolsets.

tably, SciAgent-Coder surpasses ToRA-Coder by absolute accuracy of 13.4% and 12.7% on the 7B and 13B versions. Our strongest agent, SciAgent-DeepMath-7B, substantially outperforms ChatGPT (40.0% v.s. 31.0%).

The challenges of scientific reasoning. However, our agents still lags far behind GPT-4 by absolutely 20% or even more. This gap highlights the challenges of tool-augmented scientific reasoning, as well as our benchmark.

6.4 Ablation Study

We investigate the effectiveness of components in our training data and agent modules. The specific variants we considered are as follows. (1) We remove the planning module in the agent. (2) We additionally drop the cross-retrieval strategy introduced in Section 3.2. In its place, we construct function-augmented solutions directly from $\tilde{F}_q$ and $\tilde{S}_q$. (3) We further remove all function-augmented solutions from our training data and only keep the solutions without function callings (function-free solutions). (4) We do not fine-tune agents but merely use CodeLlama as $\mathcal{M}_{\text{action}}$ for inference.

We illustrate the performance of our agents and their ablated variants in Table 3. We observe that (1) Planning module significantly improves scientific reasoning abilities. As detailed and targeted queries for the retriever, the generated plannings increase the relatedness of retrieved functions. For instance, the function's Recall@3 increases from 48.3% to 53.2% in physics domain, and from 37.3% to 39.8% in chemistry domain. (2) The use of the cross-retrieval strategy is essential. Otherwise, the function-augmented solutions directly from $\tilde{F}_q$ and $\tilde{S}_q$ degrade the performance because they are too artificial and ad-hoc to teach LLMs using functions properly. (3) The absence of function-augmented solutions results in a performance drop (row 1 v.s. row 4 in Table 3) of 5.9% and 5.3% in absolute accuracy for 7B and 13B LLMs, respectively. It underscores the critical role of function-augmented solutions to enhance LLMs' tool-use abilities, and the necessity of our MathFunc corpus. (4) The removal of function-free solutions (row 4 v.s. row 5) leads to an absolutely 14.4% accuracy decrease. Specifically focusing on non-math samples, there is a notable performance drop of about 12% as well.

Table 3: Ablation study on *human-annotated* subset of SciToolBench. We report the accuracy of samples across (1) all domains, (2) four domains excluding the math domain (wo. math).

	Planning	**Function-augmented solutions**	**Function-free solutions**	**Accuracy (7B)**		**Accuracy (13B)**	
				All	wo. math	All	wo. math
SciAgent-Coder	✓	✓(cross-retrieval)	✓	**32.2**	**34.6**	**35.7**	**36.5**
Intermediate variants 1-3	✗	✓(cross-retrieval)	✓	30.3	33.9	32.8	34.4
	✗	✓(direct-use)	✓	17.8	17.3	26.6	31.0
	✗	✗	✓	26.3	26.1	30.4	31.7
CodeLlama	✗	✗	✗	11.9	14.7	16.0	19.4

This clearly demonstrates the fundamental importance of math skills in diverse scientific reasoning tasks, and highlights how our math-related samples enhance LLMs' capabilities in this area.

6.5 Analysis[9]

Robustness of Toolsets. We acknowledge the construction and maintenance of toolsets is sometime challenging. Therefore, we stress the importance of our agents' robustness. If a sub-par toolset were provided, an robust agent should at the very least perform comparably, if not better, than other competitive LLMs without tool-use. To evaluate the robustness of SciAgent-Coder, we simulate two sub-par settings. (1) weak-related: for each question, we restrict the agents from retrieving functions that are directly derived from it. This setting greatly decreases the likelihood of retrieving a proper function from the toolset. (2) unrelated: we completely remove the domain-specific toolset in SciToolBench. As a substitution, we provide the unrelated toolset constructed in MathFunc.

Table 4: Accuracy on SciAgent with sub-par toolsets. **WR**: weak-related toolsets. **UR**: unrelated toolsets. **NA**: No toolset. The subscripts indicate the difference from the best LLMs (wo. toolsets) each column.

Model	**Toolset**	**Accuracy (7B)**		**Accuracy (13B)**	
		All	wo.math	All	wo. math
SciAgent -Coder	WR	$18.8_{+0.7}$	$18.0_{+8.3}$	$24.6_{+4.6}$	$19.9_{+7.6}$
	UR	$14.7_{-3.7}$	$10.7_{+1.0}$	$20.3_{+0.3}$	$14.7_{+2.4}$
MAmmo-C	NA	12.7	9.0	16.4	12.3
ToRA-C	NA	18.1	9.7	20.0	11.1

We compare our agents with two competitive LLMs, *i.e.,* ToRA-Coder and MAmmoTH-Coder, in above two settings. As shown in Table 4, (1) SciAgent series with unrelated toolsets present comparable performance with the two LLMs. In other words, our tool-augmented agents are unlikely to degrade the performance even under the extreme scenarios. (2) Our agents with weak-related toolsets significantly outperform the two LLMs, which further validates the robustness.

The Effect of Retriever Quality. We explore the effect of retriever quality on the ending performance. We substitute our fine-tuned retriever in SciAgent series by two competitive variants: SimCSE (Gao et al., 2021) and Contriever (Izacard et al., 2021). As shown in Figure 6 (top), our retriever surpasses the other two. It shows that fine-tuning on the math domain benefits the retrieval of tools in the generalized scientific domains.

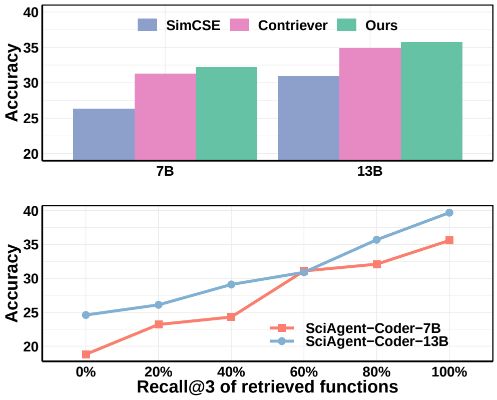


Figure 6: **Top**: Performance of SciAgent-Coder on SciToolBench with different retriever variants. **Bottom**: Relationship between the performance and the hit@3 of retrieved functions (artificially controlled).

We further dive deep into the relationship between the hit ratio of tools and the agents' performance. To this end, we manually control the hit@3 ratio by artificially adding/removing the positive functions to/from the retrieved list. Results in Figure 6 (bottom) show a clearly positive correlation between the hit ratio and the task accuracy. It illustrates that the retrieved functions facilitate the reasoning of scientific problems. However, we still observe a limit (40% accuracy) when the hit ratios

[9] We use the *human-annotated* subsets of SciToolBench for evaluations in this section. It is due to that samples in this subset have ground-truth function-augmented solutions, which are necessary for fine-grained discussion and analysis.

reaching 100%, showing the challenge of scientific reasoning even when aided by tools. We hope the future work to bridge this performance gap.

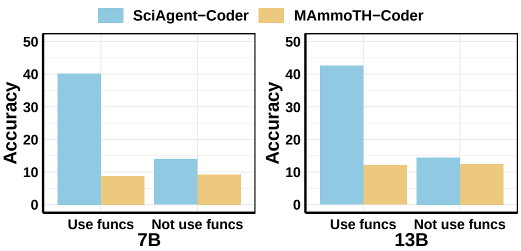


Figure 7: The performance of SCIAGENT-CODER (w. toolset) and MAmmoTH-Coder (wo. toolset) on samples which (1) use and (2) not use retrieved functions.

How the Retrieved Functions Benefit. To assess how the retrieved functions aid in the reasoning process of LLMs, we divided the samples into two subsets based on whether our agents use the retrieved functions to solve the problems. We evaluate the performance of these two subsets respectively, comparing with MAmmoTH-Coder series (without tool-use). The results in Figure 7 reveal a two-fold benefit: (1) For samples where functions are explicitly called to solve the questions, our agents demonstrate a substantial 25% improvement in absolute accuracy over LLMs that do not have access to functions. (2) Even for samples that do not directly use functions in their written program, we still observe a slight improvement. It suggests that our agents are capable of learning from retrieved functions as a reference, and then imitate these functions to write their own programs. For instance, example in Figure 13 shows the agents learn how to use `scipy.integrate` by observing the retrieved function `average_value_of_function(...)`.

7 Conclusion

This work proposes *tool-augmented* scientific reasoning, a task aiming to solve challenging scientific problems aided by generalized and scalable tools. To facilitate and evaluate the scientific tool-use abilities of LLMs, we construct a math-related, tool-augmented training corpus MATHFUNC and a benchmark SCITOOLBENCH covering 5 scientific domains. Additionally, we develop open-source agents, SCIAGENT series, as competitive baselines. Extensive experiments reveal that our agents exhibit tool-use abilities exceeding ChatGPT in scientific reasoning tasks.

Limitations

The primary limitation of our work comes from the way we compile `SciToolBench`.

Toolsets The tools are constructed directly based on the benchmark's questions, raising concerns about potential information leakage. To address this, we invest significant human effort in our annotation process as detailed in Appendix D.3. We manually review and, if necessary, revise all derived functions to ensure their generalizability and quality. As shown in Figure 6 (bottom), our agents achieve only about 40% accuracy when we provide each question the exact function from which it derives (*i.e.,* 100% hit ratio). It not only highlights the inherent challenge of scientific reasoning tasks, but also suggests that our benchmark suffers minimal impact from the potential information leakage.

Question We expand the questions in our benchmark by adding auto-synthesized questions. Though the evaluation on synthesized datasets has been commonly used by some recent work (Masry et al., 2022; Ge et al., 2024), the rationality and quality of these synthesized questions deserve deeper discussion. To this end, we compare the results on human-annotated questions and synthesized questions and observe a high correlation between them on different LLMs. Quantitatively, they have a Spearman's Rank Correlation Coefficient as 0.903 (p-value 2.2e-8). In other words, if a model performs better than another model on synthesized questions, it very likely achieves a higher score on human-annotated questions. Therefore, we believe that the accuracy on synthesized dataset could overall indicate the tool-use abilities of these LLMs.

Ethics Statement

We ensure that SCITOOLBENCH was constructed in compliance with the terms of use of all source materials and with full respect for the intellectual property and privacy rights of the original authors of the texts. We also provide details on the characteristics and annotation steps of SCITOOLBENCH in Section 5 and Appendix D. We believe our created datasets do not cause any potential risks.

References

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. 2023. Llemma: An open language model for mathematics.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. 2023. Chemcrow: Augmenting large-language models with chemistry tools.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023a. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. 2023b. TheoremQA: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7889–7901, Singapore. Association for Computational Linguistics.

Zhipeng Chen, Kun Zhou, Beichen Zhang, Zheng Gong, Xin Zhao, and Ji-Rong Wen. 2023c. ChatCoT: Tool-augmented chain-of-thought reasoning on chat-based large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14777–14790, Singapore. Association for Computational Linguistics.

Ethan Chern, Haoyang Zou, Xuefeng Li, Jiewen Hu, Kehua Feng, Junlong Li, and Pengfei Liu. 2023. Generative ai for math: Abel. `https://github.com/GAIR-NLP/abel`.

Yin Fang, Xiaozhuan Liang, Ningyu Zhang, Kangwei Liu, Rui Huang, Zhuo Chen, Xiaohui Fan, and Huajun Chen. 2023. Mol-instructions: A large-scale biomolecular instruction dataset for large language models.

Shen Gao, Zhengliang Shi, Minghang Zhu, Bowen Fang, Xin Xin, Pengjie Ren, Zhumin Chen, Jun Ma, and Zhaochun Ren. 2023. Confucius: Iterative tool learning from introspection feedback by easy-to-difficult curriculum.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. 2024. Scaling synthetic data creation with 1,000,000,000 personas.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023a. Critic: Large language models can self-correct with tool-interactive critiquing.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2023b. Tora: A tool-integrated reasoning agent for mathematical problem solving.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021a. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Jiayi Lei, Yao Fu, Maosong Sun, and Junxian He. 2023. C-eval: A multi-level multi-discipline chinese evaluation suite for foundation models.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2021. Unsupervised dense information retrieval with contrastive learning.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. 2023. Genegpt: Augmenting large language models with domain tools for improved access to biomedical information.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Zhengliang Li, Liang Chen, Yiming Zong, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2023. Ml-bench: Large language models leverage open-source libraries for machine learning tasks.

LlamaTeam. 2024. The llama 3 herd of models.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models.

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.

Ahmed Masry, Xuan Long Do, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. 2022. ChartQA: A benchmark for question answering about charts with visual and logical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2263–2279, Dublin, Ireland. Association for Computational Linguistics.

OpenAI. 2023. Gpt-4 technical report.

Siru Ouyang, Zhuosheng Zhang, Bing Yan, Xuan Liu, Jiawei Han, and Lianhui Qin. 2023. Structured chemistry reasoning with large language models.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback.

PhiTeam. 2024. Phi-3 technical report: A highly capable language model locally on your phone.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore. Association for Computational Linguistics.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023a. Tool learning with foundation models.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023b. Toolllm: Facilitating large language models to master 16000+ real-world apis.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA. Association for Computing Machinery.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. In *Advances in Neural Information Processing Systems*.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world restful apis.

Liangtai Sun, Yang Han, Zihan Zhao, Da Ma, Zhennan Shen, Baocai Chen, Lu Chen, and Kai Yu. 2023. Scieval: A multi-level large language model evaluation benchmark for scientific research.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti

Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2019. Representation learning with contrastive predictive coding.

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

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R. Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. 2023b. Scibench: Evaluating college-level scientific problem-solving abilities of large language models.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2023c. Mint: Evaluating llms in multi-turn interaction with tools and language feedback.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023a. On the tool manipulation capability of open-source large language models.

Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, Zhoujun Cheng, Siheng Zhao, Lingpeng Kong, Bailin Wang, Caiming Xiong, and Tao Yu. 2023b. Lemur: Harmonizing natural language and code for language agents.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. Gpt4tools: Teaching large language model to use tools via self-instruction.

Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. 2023. Lumos: Learning agents with unified data, modular design, and open-source llms.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. *ArXiv preprint*, abs/2309.12284.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. 2023a. Craft: Customizing llms by creating and retrieving from specialized toolsets.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023b. Scaling relationship on learning mathematical reasoning with large language models.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2023a. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi.

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

Dan Zhang, Ziniu Hu, Sining Zhoubian, Zhengxiao Du, Kaiyu Yang, Zihan Wang, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. Sciglm: Training scientific language models with self-reflective instruction annotation and tuning.

Wenxuan Zhang, Sharifah Mahani Aljunied, Chang Gao, Yew Ken Chia, and Lidong Bing. 2023a. M3exam: A multilingual, multimodal, multilevel benchmark for examining large language models.

Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. 2023b. Cumulative reasoning with large language models.

Yilun Zhao, Hongjun Liu, Yitao Long, Rui Zhang, Chen Zhao, and Arman Cohan. 2023. Knowledgemath: Knowledge-intensive math word problem solving in finance domains.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification.

A Detailed Related Work

A.1 Scientific Reasoning

Scientific reasoning can be roughly categorized into two branches: (1) mathematical reasoning and (2) reasoning across other scientific domains.

Mathematical Reasoning. As a fundamental task to evaluate the capabilities of LLMs (PhiTeam, 2024; LlamaTeam, 2024), mathematical (math) reasoning has attracted much more attentions recently. There are intensive studies for more powerful math-oriented LLMs by prompt engineering (Qian et al., 2023; Zhang et al., 2023b; Zhou et al., 2023), instruction-tuning (Yuan et al., 2023b; Yue et al., 2023b; Gou et al., 2023b; Yu et al., 2023; Wang et al., 2023a) and even pre-training (Luo et al., 2023; Azerbayev et al., 2023; Chern et al., 2023). Regarding instruction-tuning, we notice that recent studies have automatically constructed high-quality instructions from GPT-4, *i.e.,* fine-tuning open-source LLMs by Program-of-thought (PoT; Chen et al. 2023a) prompting. It enables open-source LLMs to present remarkable performance, even comparable with GPT-4.

Reasoning across Other Domains. There have been intensive works on scientific LLMs (Bran et al., 2023; Jin et al., 2023; Fang et al., 2023) and benchmarks (Hendrycks et al., 2021a; Huang et al., 2023; Zhang et al., 2023a; Yue et al., 2023a; Sun et al., 2023). However, they primarily target on problems involving less complicated reasoning like knowledge retrieval or simple tool utilization.

Regarding complicated scientific reasoning problems (Chen et al., 2023b; Wang et al., 2023b), questions are scattered among diverse topics and each topic additionally requires domain-specific knowledge. So annotating questions and their solutions domain by domain is much more labor-consuming. Most current benchmarks (Chen et al., 2023b; Wang et al., 2023b; Zhao et al., 2023) merely include hundreds of questions (in all; less for each single domain) from textbooks and provide no training samples. A concurrent work (Zhang et al., 2024) develop a large-scale scientific training corpus, but only focuses three common domains: math, physical and chemistry. Accordingly, the progress of reasoning tasks in these domains is slower than that in math domain: the most competitive approach only achieves 50% and 35% on TheoremQA and SciBench, respectively. Instead of developing an omniscient and proficient LLMs on reasoning tasks across various scientific domains, we believe it is more practical to teach LLMs the ability to use domain-specific tools to facilitate their reasoning abilities in some domain when external functions (toolset) are attached.

A.2 Tool Learning

LLMs, both proprietary ones and open-source ones, demonstrate promising capabilities leveraging external tools to solve problems beyond their limits (Qin et al., 2023a). Combined with specific tools, these *tool-augmented* LLMs achieve great success on various tasks such as machine learning (Wu et al., 2023; Shen et al., 2023; Patil et al., 2023; Yang et al., 2023; Liu et al., 2023), question answering (Peng et al., 2023; Gou et al., 2023a), daily assistance (Xu et al., 2023a; Qin et al., 2023b; Song et al., 2023; Gao et al., 2023), *etc.*

Previous work usually pre-defines several tools, *e.g.,* equation solver or calculator, to facilitate math reasoning tasks (Gou et al., 2023a; Lu et al., 2023; Hao et al., 2023; Chen et al., 2023c; Wang et al., 2023c; Xu et al., 2023b; Yin et al., 2023). Cai et al. (2023) generalize the concept of tools to *Program functions*. Following this concept, CREATOR (Qian et al., 2023) scale up the function number towards thousand level. However, these ad-hoc, argument-free functions are more like solution wrapper rather than well-generalized tools. CRAFT (Yuan et al., 2023a) targetedly design an automatic pipeline to extract generalized functions for tool-use. Though leading to improvement, these functions are still not generalized enough and serve more as reference rather than as tools for direct calling [10]. Ouyang et al. 2023 ask LLM to generate chemistry formulae as knowledge reference to assist the following reasoning and achieve enhanced performance on chemistry questions in SciBench. Similar as our attached toolset, Zhao et al. (2023) maintain a *knowledge bank* in which saves more than 900 financial definitions/equations/models as the format of functions for retrieval and use. To our best knowledge, our work is the first which (1) fine-tunes open-source, tool-augmented LLM agents for scientific reasoning tasks and (2) provides a benchmark covering multiple scientific domains to evaluate LLMs' tool-use abilities.

[10] We check CRAFT's results on MATH test set under their official repository. We observe that only 172 out of 880 test instances (19.5%) explicitly call the retrieved functions for the problem solving. Since the performance improvement shall be attributed more to reference than explicit calling, we speculate that the created functions are still not generalized enough.

B Training Details

B.1 Retriever

To fine-tune a retriever, we construct the training samples from MathFunc. We concatenate the question and its planning as the query, and view the generated functions as the keys. We finally collect a total of 8,603 query-key pairs for training, and split 10% training samples as validation set.

$$\text{query} = [q; G_q]$$
$$\text{key} = f \in \tilde{F}_q$$

We follow DPR (Karpukhin et al., 2020) to train a dense retriever R. We use RoBERTa-base (Liu et al., 2019) as the backbone. We set the training step as 500, the batch size as 128 and the learning rate as 2e-5. We also set the temperature coefficient of the InfoNCE loss (van den Oord et al., 2019) as 0.07. We run this experiment on a single NVIDIA Quadro RTX8000 GPU. The whole training process lasts for about 20 minutes.

B.2 Planning and Action

We fine-tune CodeLlama (Rozière et al., 2023), Mistral (Jiang et al., 2023), Llama-3 (Touvron et al., 2023) and DeepMath (Shao et al., 2024) on MathFunc to develop the planning and action modules in our tool-augmented agents SciAgent series. We set the global batch size as 128. We use the learning rate as 2e-5 for CodeLlama, 2e-6 for Mistral and Llama-3, and 5e-6 for DeepMath. We use a cosine scheduler with a 3% warm-up period for 2 epochs. We train all models with *ZeRO Stage3* (Rajbhandari et al., 2021) on 8 V100 GPUs. The whole training process lasts for about 3 hours for 7B LLMs and 7.5 hours for 13B LLMs.

The planning and action modules share the same model but act differently with different input instructions. We detail the format of planning and action instructions as below:

Planning. Given a question q, we construct a planning sample as $(I_{\text{plan}}(q), G_q)$, where $I_{\text{plan}}(q)$ is the input instruction, G_q is the output, and $I_{\text{plan}}(.)$ is the template for planning module. We provide an example of planning instruction as below:

Listing 1: An example of the planning sample. We separate the input instruction and output answer by the dashed line.

```
Read the following question and provide
    a high-level, step-by-step plan for
    this problem.

Question: Two complementary angles are
    in a ratio of $3:2$. What is the
    measure, in degrees, of the smaller
    angle?

Answer:
-----------------------------------------
To solve this problem, we need to
    perform the following steps:
- Set up an equation using the given
    ratio and the fact that the angles
    are complementary.
- Solve the equation for the variable
    representing the ratio.
- Calculate the measure of the smaller
    angle using the found variable value.
```

Action. Given a question q, we construct an action sample as $(I_{\text{action}}(q, F_q), S_q)$, where $I_{\text{action}}(q, F_q)$ is the input instruction, S_q is the output answer, and $I_{\text{action}}(.)$ is the template for action module. F_q are retrieved functions from the toolset in MathFunc. We adopt the retriever in Appendix B.1 to retrieve three functions in the toolset. We provide an example of action instruction as below:

Listing 2: An example of the action sample. We separate the input instruction and output answer by the dashed line. We only show one retrieved function in this sample for visualization convenience.

```
Read the following mathematical question
     and answer it. Please note that you
     could (optionally, not required)
    call the following functions when
    writing the code:

def angle_ratio_to_degrees(ratio):
   """
   Computes the angle measures in
       degrees given their ratio.

   Parameters:
   - ratio (tuple): The ratio of the
       angles.

   Returns:
   - tuple: The angle measures in
       degrees.

   Formula:
   - angle_1 = ratio[0] * x
   - angle_2 = ratio[1] * x
   - angle_3 = ratio[2] * x
   - where x is the common variable such
        that angle_1 + angle_2 + angle_3
        = 180 degrees
   """
   sum_ratios = sum(ratio)
   x = 180 / sum_ratios
   return tuple(angle * x for angle in
       ratio)

[...Two more retrieved functions...]
```

```
Question: Two complementary angles are
    in a ratio of $3:2$. What is the
    measure, in degrees, of the smaller
    angle?

Answer:
---------------------------------------
# Define the ratio of the angles.
ratio = (3, 2)
# Since the angles are complementary,
    they add up to 90 degrees.
# We can find the value of each part of
    the ratio by dividing 90 by the sum
    of the parts of the ratio.
value_of_each_part = 90 / sum(ratio)
# The measure of the smaller angle is
    the value of each part times the
    smaller part of the ratio.
smaller_angle = value_of_each_part * min
    (ratio)
# Print the result.
print(smaller_angle)
```

C Evaluation Details

We evaluate (1) the tool-use abilities of LLMs in scientific reasoning task, and (2) their performance when supplemented without any toolsets for comparison. We detail these two settings as below:

With Toolsets. We use the unified PoT-based prompt (Chen et al., 2023a) for all pretraining-based models and our SciAgent series. The unified prompt consists of a short task description and two demonstrations. We show the prompt in Appendix H.4. For each question, we provide three retrieved functions and instruct LLMs to use them if (and only if) necessary. Note that we use the same retriever, *i.e.,* fine-tuned from MathFunc, for all LLMs. For MAmmoTH-Coder and ToRA-Coder which are fine-tuned on specific (tool-agnostic) instructions, we try to enable them to use retrieved tools while keeping the formats of their original instructions as much as possible. Specifically, we append a short *tool-augmented* description at the end of their original prompts:

```
[original prompt]

Please note that you could (optionally,
    not required) call the following
    functions when writing the program:

[retrieved functions]
```

Without Toolsets. Similar as above, we use the unified PoT-based prompt (Chen et al., 2023a) shown in Appendix H.5 for all pretraining-based models and our SciAgent series. And we follow the original instructions used for MAmmoTH-Coder and ToRA-Coder to evaluate their performance.

D Details of SciToolBench Annotation

We provide a more thorough description about SciToolBench construction in this section. This semi-automatic annotation pipeline involves both GPT-4 and humans to balance the quality and cost. Specifically, we enlist two authors to serve as human annotators. Both of them are graduate students with proficiency in English. Additionally, they hold Bachelor of Science and/or Engineering degrees and have completed undergraduate-level courses in the five scientific domains corresponding to our benchmark. We detail the four subsequent submodules in our annotation pipeline, *i.e.,* human-annotated question curation, synthesized question generation, positive function construction and negative function construction, as below.

D.1 Human-annotated Question Curation

We curate the questions from TheoremQA (Chen et al., 2023b) and SciBench (Wang et al., 2023b), both of which are available under the MIT License. Among 1,495 questions in these original two datasets, we remove three kinds of questions.

Image-required: There are 37 questions from TheoremQA which include images and necessitate visual understanding abilities. We remove these samples because our benchmark is text-oriented.

Reasoning-agnostic: There are some multi-choice questions from TheoremQA which merely requires the memorization of knowledge points but involves little reasoning process. For example:

> **Question:** The open mapping theorem can be proved by
> (a) Baire category theorem.
> (b) Cauchy integral theorem.
> (c) Random graph theorem.
> (d) None of the above.

We manually check each samples and remove 68 such kind of samples.

Over-difficult: Too hard questions confuse all models and weaken the discrimination of our benchmark. To balance the difficulty and discrimination, we employ 4 advanced proprietary models [11] to generate related functions and function-augmented program solutions. We generate 6 solutions for each model (one generated by greedy decoding and the other five by nucleus sampling

[11] gpt-4, gpt4-32k, gpt-3.5-turbo, gpt-3.5-turbo-16k

with 0.6 temperature) and 24 solutions in all. We view questions that are answered incorrectly by all 24 solutions as *over-difficult* questions. We remove all *over-difficult* questions, and retain 73.5% questions in TheoremQA and 47.8% in SciBench.

By removing three kinds of samples mentioned above, there are a total of 856 questions in our SciToolBench benchmark.

D.2 Synthesized Question Generation

The *human-annotated* questions mentioned above are curated from two small-scale yet diverse datasets. As a result, there are limited questions that share the same knowledge points and, consequently, the functions in the toolsets. This limitation may constrain the validation of the toolset's generalizability. To address this gap, we expand the question set for better varied applicability of the functions. We synthesize new questions by a two-step, automatic pipeline as below:

Question Generation. For each human-annotated question, we employ GPT-4o to generate an additional six *similar but not identical* questions. To ensure that the generated questions are as independent as possible, we (1) set a high temperature of 1.0, and (2) run GPT-4o six times in parallel, generating one question in each run to prevent the influence of previously generated questions on new ones. We show the used prompt as below.

Listing 3: Prompt for synthesized question generation

~~~
Given a scientific question, you are
    tasked to generate a new question
    following the requirements as below:
- The new question should be
    quantitative, i.e., the answer is a
    specific number.
- The new question should share the same
     scientific core knowledge or
    formula as the original one.
- The new question should not be too
    similar to the original question.
    You should make significant changes
    to the question by altering the
    narrative, context, specific numbers,
     etc.
- The background and settings of the new
     question should be realistic,
    avoiding any impossible scenarios
    such as a 2000kg human or a
    temperature of -100K.

Output format:
```Question
[New Question]
```
~~~

Question Filtering. The above step results in the generation of 5,136 synthesized questions WITHOUT ground-truth answers and function-augmented solutions. To ensure the quality of these questions, we have GPT-4o generate answers five times for each synthesized question and adopt the majority vote as the (silver) answer. Questions for which GPT-4o fails to provide a major-voting answer are removed. Additionally, a manual review of some synthesized questions reveals that GPT-4o occasionally generates *overly simplistic* questions, which diminishes the necessity for tool-use applications. Therefore, questions with unanimously predictions (5/5) are considered *overly simplistic* and downsampled at a ratio of 0.5. Consequently, a total of 3,394 out of 5,136 synthesized questions are finalized as part of our benchmark.

D.3 Positive Function Construction

Function Generation

In practice, we merge this sub-module to the process of over-difficult question identification in Appendix D.1. We randomly sample one set of functions which yield correct solutions for each question. As a result, we collect a total of 1,216 candidates for the next verification sub-module. We additionally save other functions leading to correct solutions and use them as reference in the refinement sub-module.

Function Verification

We verify the generated functions from both correctness and generalizations. We detail them separately as below.

1. Correctness: Since all candidate functions lead to correct solutions, we speculate that almost all of them are correct. We randomly sample 100 functions (20 per domain) and manually check their correctness. The results shown in Table 5 validate our speculation. Therefore, we assume all candidate functions are correct and retain them.

Table 5: The correctness of 100 randomly sampled functions across five domains.

	Correct	Partially Correct	Wrong	All
Math	18	2	0	20
Physics	19	1	0	20
Chemistry	20	0	0	20
Finance	19	0	1	20
EECS	17	3	0	20
All	93	6	1	100

2. Generalization: We encounter the similar problem as the function construction in MathFunc, *i.e.,* some of the auto-generated functions are not generalized enough. If ad-hoc functions were in the provided toolsets of our benchmark, they would cause a significant overestimation of LLMs' tool-use abilities. To mitigate it as much as possible, we manually check all candidate functions to ensure their generalization. Specifically, we design a binary classification task and assign each function a label in {`Retained`, `Refined`}. We label a function as `refined` if it had one of the problems listed below: (1) a pure solution wrapper. (2) merely defining a non-generalized expression (likely only occur in this question). (3) the argument names or document describing the special scenario of corresponding question and not being generalized/abstractive enough. (4) including ad-hoc constants or code snippets. The annotators firstly co-annotate 100 functions. We calculate Cohen's kappa value of their annotation results as 0.85, illustrating an ideal agreement. Therefore, the annotators separately annotate the remaining functions. It takes about 6 hours per annotator to classify about 650 functions. We show some `Refined` function cases in Figure 11, and the annotation interface in Figure 9.

As a result, we collect 1,012 `Retained` and 206 `Refined` functions. We keep all `Retained` as the component of positive functions. We also feed the `Refined` functions to next refinement sub-module to modify them as much as possible.

Function Refinement

This sub-module aims to rewrite 206 `Refined` functions to make them qualified. To this end, we associate each function with (1) the question from which it is derived, (2) the function-augmented solutions, and (3) the alternative functions from the generation sub-module (if have). Then we provide them to the annotators. The annotators are asked to rewrite the functions to **improve their generalization** as much as possible. If one function were successfully rewritten, we also require the annotator to write a solution involving the new function to the related question. The solution must yield correct answer to ensure the correctness of the rewritten function. We show some rewritten cases in Figure 11, and the screenshot of the annotation interface in Figure 10.

It takes approximately 12 hours per annotator to check each `Refined` function and, if applicable, rewrite it. As a consequence, we successfully rewrite 91 `Refined` functions and drop the remaining ones. We combine these 91 rewritten functions and the 1,012 `Retained` functions to construct 1,103 positive functions. At last step, we deduplicate these functions and finalize a collection of 942 functions.

D.4 Negative Function Construction

The positive functions constructed above have satisfied the minimum requirements of the toolset in our benchmark. However, we find that such kind of benchmark contains shortcuts for LLM to retrieve and use functions. Take a physical question about *frequency-angular conversion* as example, the previous modules construct a positive function named `angular_from_frequency`(...) to solve this question. Without any other similar functions, the LLMs could readily select and use the **only** function by superficial shortcuts. These shortcuts significantly weaken the function-understanding and -use abilities evaluation of our benchmark. To mitigate this problem, we design an additional module to eliminate the shortcuts by constructing some (hard) negative functions for each positive function, like `frequency_from_angular`(...) and `frequency_from_energy`(...) in the above example. Among three similar functions, LLMs are forced to understand their usages and choose proper ones to use. In summary, we add negative functions into the toolset to simulate a more challenging scenario and better evaluate LLMs' tool-use abilities.

Listing 4: Prompt for constructing negative functions

~~~
Given a function about the {subfield}
    field, could you please write two
    more functions which satisfy:
- The functions should be in the same
    field with the provided function,
    while the knowledge point is not
    compulsorily the same.
- The functions should be similar, but
    not identical with the provided
    function.
- The new written functions should be
    wrapped as the below format:

New function 1:
```python
[new_written_function_1]
```

New function 2:
```python
[new_written_function_2]
```
~~~

Specifically, we employ GPT-4 for each positive function to generate two similar but not identical functions as the negative functions. The prompt used is shown as below. We do not validate the correctness of negative functions for simplicity, as they are not intended to be used for any question. We filter the duplicated functions and retain the other 1,343 functions in all. By merging 942 positive functions and 1,343 negative functions, we finally collect a total of 2,285 functions in our toolset.

E More Details for Datasets

E.1 MathFunc

Quality Checking We conduct quality checking on the functions and solutions in MathFunc. Specifically, we randomly sample 100 functions from the toolset and 100 function-augmented solutions from the sample set[12]. Then we manually check the correctness of them as what we have done in Appendix D.3. The results demonstrate that 96 functions and 98 solutions are correct, which validate the quality of these auto-generated data and our construction pipeline.

More Statistics We count the number of used functions in each function-augmented solution, *i.e.,* the function occurrence, and show the results as below.

Function Occurrence	Count
0	1250
1	712
2	91
3	20
4	18
≥ 5	10

Table 6: Function occurrence in MathFunc

We find that (1) 40.3% of solutions do not call any functions. We deliberately include these samples in MathFunco enhance the model's robustness, *i.e.,* learning not to use retrieved functions if they were not appropriate. (2) For other solutions, each of them calls 1.31 functions on average.

Function Examples We list the top-10 frequent functions and other 10 representative high-frequency functions in the toolset of MathFunc in the following two tables.

Function name	Frequency	Function name	Frequency
`combinations`	80	`solve_quadratic`	44
`gcd`	64	`triangle_area`	43
`factorial`	58	`solve_quadratic`	40
`is_prime`	52	`circle_area`	38
`solve_linear_system`	51	`binomial_coefficient`	37

Table 7: Top-10 frequent functions in MathFunc

Function name	Frequency	Function name	Frequency
`mod_exp`	24	`repeating_decimal_to_fraction`	14
`base_n_to_base_10`	19	`dot_product`	14
`degrees_to_radians`	17	`arrangements_with_repeats`	11
`is_palindrome`	15	`arithmetic_sequence_nth_term`	9
`simplify_expression`	15	`sum_of_arithmetic_sequence`	9

Table 8: Ten representative functions in MathFunc

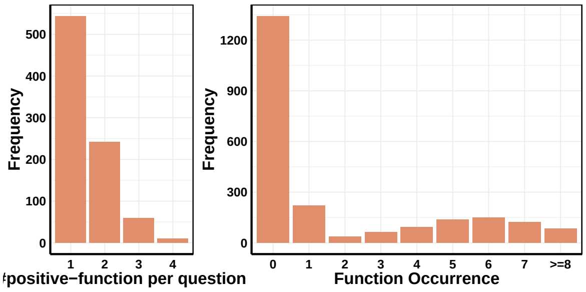


Figure 8: **Left**: Histogram of FPQ (function per question). Higher values indicate greater composability. **Right**: Histogram of function occurrence. Higher values indicate more generalization and wider application.

E.2 SciToolBench

More Statistics We present additional statistics to illustrate the composability and generalization of the toolsets in SciToolBench. (1) Regarding composability, we count the FPQ (the number of positive functions used in solutions) for each question in the human-annotated subset, as shown in Figure 8 (left). The results indicate that more than 36% questions call more than one functions in their golden solutions, and each question requires an average of 1.51 functions. They demonstrate a high degree of composability on these functions. (2) For generalization, we provide the statistics about the functions' usage frequency among the whole question set in Figure 8 (right) [13]. We observe a clear bi-modal distribution pattern, which we attribute to the presence of negative functions. As explained in Appendix D.4, we include a number of negative functions, *i.e.,* functions that are never used by any questions and are counted as 0 in the aforementioned figure, in our toolset to eliminate

[12] We assume the correctness of their called functions when we check the solutions.

[13] Although there are no golden function-augmented solutions for the synthesized subset, we estimate the function occurrences using the following approximation: *The synthesized questions inherit the human-annotated functions from which they are derived.* We argue that this approximation is reasonable because the synthesized questions share the same knowledge points as the original human-annotated questions. Additionally, the experimental results in Figure 2 clearly demonstrate that our SciAgent series successfully utilize functions to improve performance on the synthesized subsets, validating the applicability of these functions for the synthesized questions.

Model (7B)	**Accuracy**
MAmmoTH-Coder	32.1
ToRA-Coder$_{\text{wo. output shaping}}$	40.2
ToRA-Coder	44.6
SciAgent-Coder	41.0
Model (13B)	**Accuracy**
MAmmoTH-Coder	36.3
ToRA-Coder$_{\text{wo. output shaping}}$	44.6
ToRA-Coder	48.1
SciAgent-Coder	45.2

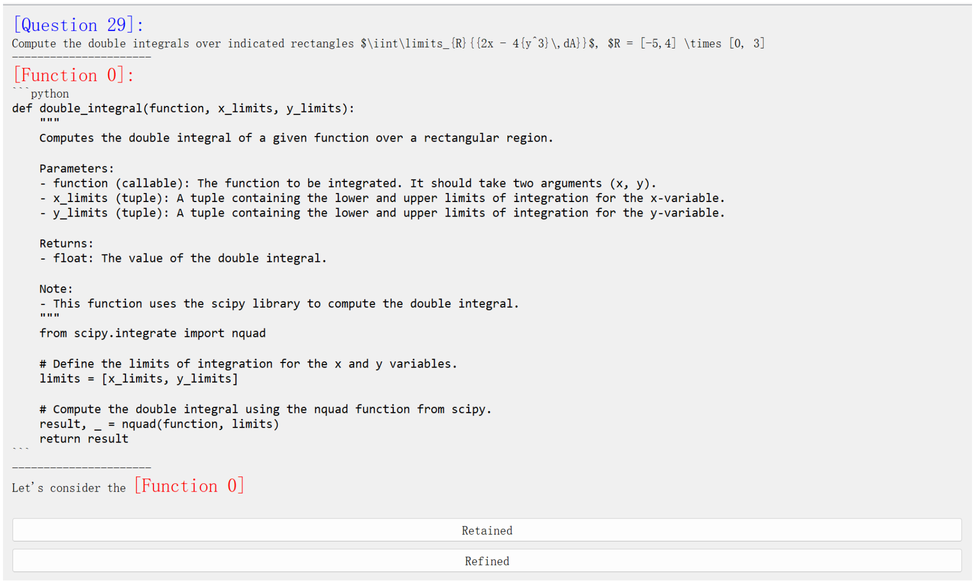
Table 9: Performance comparison on MATH test set.

the potential shortcuts. While these negative functions appear to reduce the average function occurrences, they enhance the overall generalization of our toolsets. When excluding these negative functions, the average function occurrences rise to 4.58, with over 76% of positive functions being reused. These results validate the robust generalization of our toolsets in SciToolBench.

F Discussion on In-domain Tool Using

This work facilitates LLMs' scientific reasoning abilities with the aid of tools. Due to the scarce annotations across scientific domain, we construct our training corpus, *i.e.,* MathFunc, from math domain. Here raises a natural question: *whether our fine-tuned tool-augmented agents improve the in-domain performance?*

To answer this question, we run experiments on MATH test set and show their results in Table 9. It demonstrates that our SciAgent-Coder surpasses MAmmoTH-Coder and achieves comparable performance with ToRA-Coder. However, we also do not observe significant benefit from tool augmentation on MATH test set. Though previous and concurrent work (Qian et al., 2023; Yuan et al., 2023a; Wang et al., 2024) have developed impressive tool-augmented approaches to enhance various kinds of reasoning tasks on models without additional fine-tuning, it is still an open question that whether tools benefit fine-tuned, in-domain models (especially when the tools are derived from the fine-tuned annotations). And our primary experiments here implicit that the answer might be No. We believe this question deserves deeper investigation as a future work.



[Question 29]:

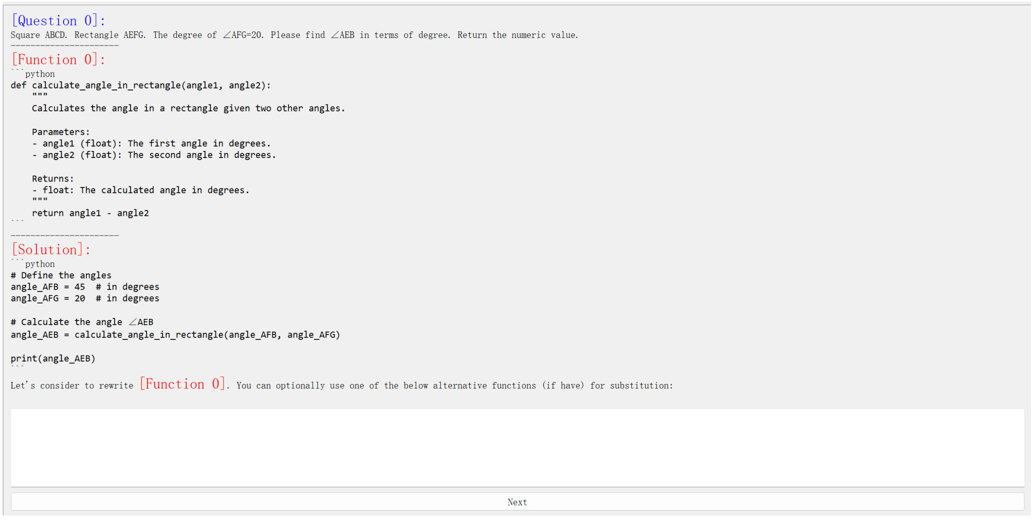
Compute the double integrals over indicated rectangles $\iint\limits_{R}{{2x - 4{y^3}\,dA}}$, $R = [-5,4] \times [0, 3]

[Function 0]:

~~~
```python
def double_integral(function, x_limits, y_limits):
    """
    Computes the double integral of a given function over a rectangular region.

    Parameters:
    - function (callable): The function to be integrated. It should take two arguments (x, y).
    - x_limits (tuple): A tuple containing the lower and upper limits of integration for the x-variable.
    - y_limits (tuple): A tuple containing the lower and upper limits of integration for the y-variable.

    Returns:
    - float: The value of the double integral.

    Note:
    - This function uses the scipy library to compute the double integral.
    """
    from scipy.integrate import nquad

    # Define the limits of integration for the x and y variables.
    limits = [x_limits, y_limits]

    # Compute the double integral using the nquad function from scipy.
    result, _ = nquad(function, limits)
    return result
```
~~~

Let's consider the [Function 0]

Figure 9: The screenshot of our annotation interface to evaluate functions' generalization.



[Question 0]:

Square ABCD. Rectangle AEFG. The degree of ∠AFG=20. Please find ∠AEB in terms of degree. Return the numeric value.

[Function 0]:

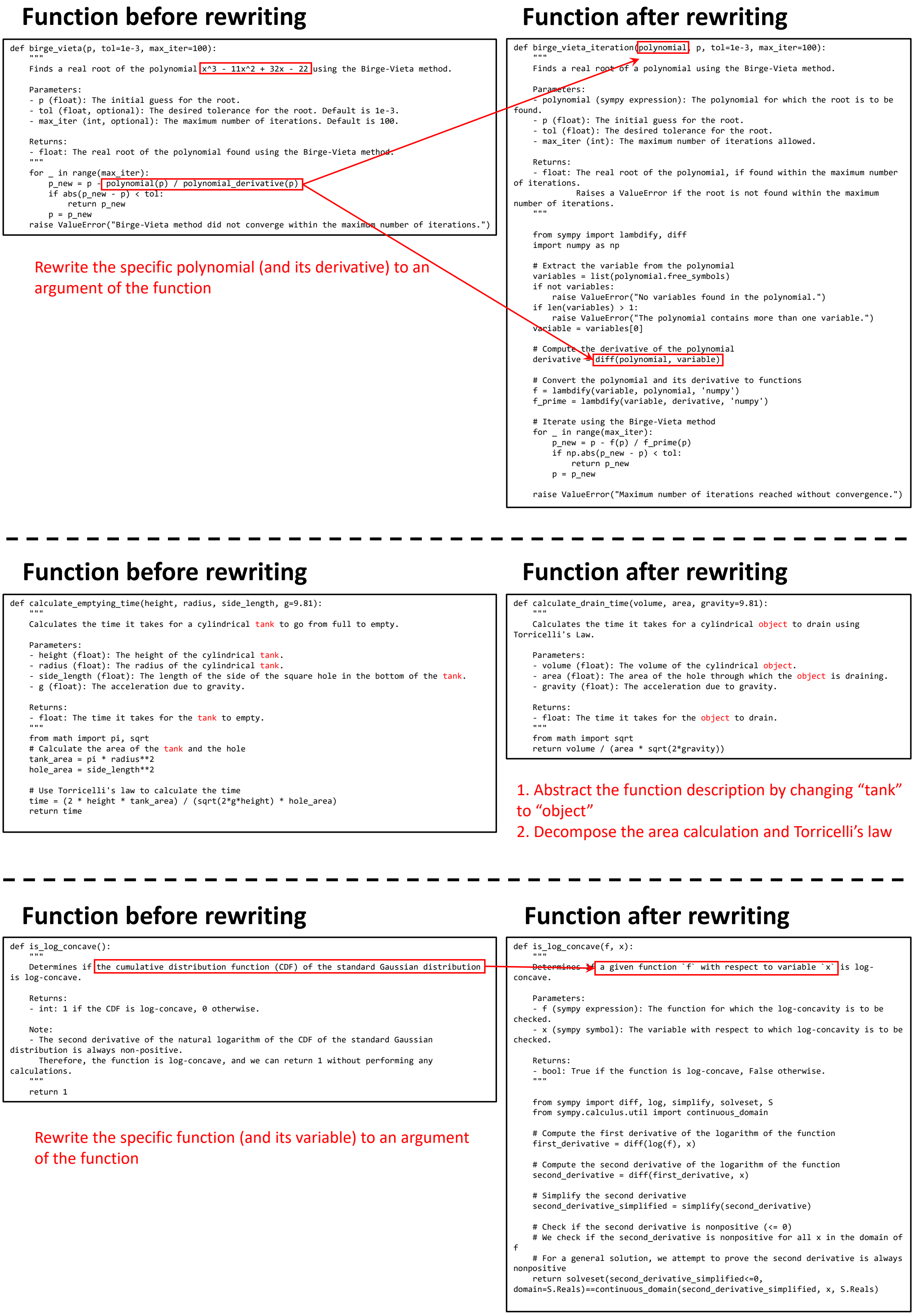
~~~
```python
def calculate_angle_in_rectangle(angle1, angle2):
    """
    Calculates the angle in a rectangle given two other angles.

    Parameters:
    - angle1 (float): The first angle in degrees.
    - angle2 (float): The second angle in degrees.

    Returns:
    - float: The calculated angle in degrees.
    """
    return angle1 - angle2
```
~~~

[Solution]:

~~~
```python
# Define the angles
angle_AFB = 45  # in degrees
angle_AFG = 20  # in degrees

# Calculate the angle ∠AEB
angle_AEB = calculate_angle_in_rectangle(angle_AFB, angle_AFG)

print(angle_AEB)
```
~~~

Let's consider to rewrite [Function 0]. You can optionally use one of the below alternative functions (if have) for substitution:

Figure 10: The screenshot of our annotation interface to rewrite functions. We provide no alternative functions in this example for convenience of visualization.



Function before rewriting

```
def birge_vieta(p, tol=1e-3, max_iter=100):
    """
    Finds a real root of the polynomial x^3 - 11x^2 + 32x - 22 using the Birge-Vieta method.

    Parameters:
    - p (float): The initial guess for the root.
    - tol (float, optional): The desired tolerance for the root. Default is 1e-3.
    - max_iter (int, optional): The maximum number of iterations. Default is 100.

    Returns:
    - float: The real root of the polynomial found using the Birge-Vieta method.
    """
    for _ in range(max_iter):
        p_new = p - polynomial(p) / polynomial_derivative(p)
        if abs(p_new - p) < tol:
            return p_new
        p = p_new
    raise ValueError("Birge-Vieta method did not converge within the maximum number of iterations.")
```

Rewrite the specific polynomial (and its derivative) to an argument of the function

Function after rewriting

```
def birge_vieta_iteration(polynomial, p, tol=1e-3, max_iter=100):
    """
    Finds a real root of a polynomial using the Birge-Vieta method.

    Parameters:
    - polynomial (sympy expression): The polynomial for which the root is to be
found.
    - p (float): The initial guess for the root.
    - tol (float): The desired tolerance for the root.
    - max_iter (int): The maximum number of iterations allowed.

    Returns:
    - float: The real root of the polynomial, if found within the maximum number
of iterations.
             Raises a ValueError if the root is not found within the maximum
number of iterations.
    """

    from sympy import lambdify, diff
    import numpy as np

    # Extract the variable from the polynomial
    variables = list(polynomial.free_symbols)
    if not variables:
        raise ValueError("No variables found in the polynomial.")
    if len(variables) > 1:
        raise ValueError("The polynomial contains more than one variable.")
    variable = variables[0]

    # Compute the derivative of the polynomial
    derivative = diff(polynomial, variable)

    # Convert the polynomial and its derivative to functions
    f = lambdify(variable, polynomial, 'numpy')
    f_prime = lambdify(variable, derivative, 'numpy')

    # Iterate using the Birge-Vieta method
    for _ in range(max_iter):
        p_new = p - f(p) / f_prime(p)
        if np.abs(p_new - p) < tol:
            return p_new
        p = p_new

    raise ValueError("Maximum number of iterations reached without convergence.")
```

Function before rewriting

```
def calculate_emptying_time(height, radius, side_length, g=9.81):
    """
    Calculates the time it takes for a cylindrical tank to go from full to empty.

    Parameters:
    - height (float): The height of the cylindrical tank.
    - radius (float): The radius of the cylindrical tank.
    - side_length (float): The length of the side of the square hole in the bottom of the tank.
    - g (float): The acceleration due to gravity.

    Returns:
    - float: The time it takes for the tank to empty.
    """
    from math import pi, sqrt
    # Calculate the area of the tank and the hole
    tank_area = pi * radius**2
    hole_area = side_length**2

    # Use Torricelli's law to calculate the time
    time = (2 * height * tank_area) / (sqrt(2*g*height) * hole_area)
    return time
```

Function after rewriting

```
def calculate_drain_time(volume, area, gravity=9.81):
    """
    Calculates the time it takes for a cylindrical object to drain using
Torricelli's Law.

    Parameters:
    - volume (float): The volume of the cylindrical object.
    - area (float): The area of the hole through which the object is draining.
    - gravity (float): The acceleration due to gravity.

    Returns:
    - float: The time it takes for the object to drain.
    """
    from math import sqrt
    return volume / (area * sqrt(2*gravity))
```

1. Abstract the function description by changing “tank” to “object”
2. Decompose the area calculation and Torricelli’s law

Function before rewriting

```
def is_log_concave():
    """
    Determines if the cumulative distribution function (CDF) of the standard Gaussian distribution
is log-concave.

    Returns:
    - int: 1 if the CDF is log-concave, 0 otherwise.

    Note:
    - The second derivative of the natural logarithm of the CDF of the standard Gaussian
distribution is always non-positive.
      Therefore, the function is log-concave, and we can return 1 without performing any
calculations.
    """
    return 1
```

Rewrite the specific function (and its variable) to an argument of the function

Function after rewriting

```
def is_log_concave(f, x):
    """
    Determines if a given function `f` with respect to variable `x` is log-
concave.

    Parameters:
    - f (sympy expression): The function for which the log-concavity is to be
checked.
    - x (sympy symbol): The variable with respect to which log-concavity is to be
checked.

    Returns:
    - bool: True if the function is log-concave, False otherwise.
    """

    from sympy import diff, log, simplify, solveset, S
    from sympy.calculus.util import continuous_domain

    # Compute the first derivative of the logarithm of the function
    first_derivative = diff(log(f), x)

    # Compute the second derivative of the logarithm of the function
    second_derivative = diff(first_derivative, x)

    # Simplify the second derivative
    second_derivative_simplified = simplify(second_derivative)

    # Check if the second derivative is nonpositive (<= 0)
    # We check if the second_derivative is nonpositive for all x in the domain of
f
    # For a general solution, we attempt to prove the second derivative is always
nonpositive
    return solveset(second_derivative_simplified<=0,
domain=S.Reals)==continuous_domain(second_derivative_simplified, x, S.Reals)
```

Figure 11: Three examples of `Refined` functions (before rewriting) and their rewritten functions (after rewriting). We also briefly describe the modifications for each example (color in red).

G Examples

Problem & Planing

```
Read the following mathematical question and answer it. Please note that you could (optionally, not
required) call the following functions when writing the code:

Question:Suppose a stock has the following information. It is listed on the London stock exchange and
operates throughout Europe. The yield on a UK 10 year treasury is 2.8%. The stock in question will earn
8.6% as per historical data. The Beta for the stock is 1.4, i.e., it is 140% volatile to the changes in
the general stock market. What is the expected rate of return?
```

```
    (1) Calculate the expected return based on the beta and the yield on a UK 10 year treasury.
    (2) Add the stock's historical return to the expected return.
```

Retrieved Functions

```python
def expected_return(rf, beta, rm):
    """
    Computes the expected return using the Capital Asset Pricing Model (CAPM) formula.

    Parameters:
    - rf (float): The risk-free rate.
    - beta (float): The beta of the portfolio.
    - rm (float): The return on the market.

    Returns:
    - float: The expected return.
    """
    return rf + beta * (rm - rf)

def expected_stock_return(true_prob, u, d):
    """
    Calculates the expected return of the stock in a one-period binomial model.

    Parameters:
    - true_prob (float): The true probability of the stock price going up.
    - u (float): One plus the rate of capital gain on the stock if the price goes up.
    - d (float): One plus the rate of capital loss on the stock if the price goes down.

    Returns:
    - float: The expected return of the stock.
    """
    return true_prob * u + (1 - true_prob) * d

...
```

Solution

```python
# Given values.
rf = 0.028  # The yield on a UK 10 year treasury
beta = 1.4  # The stock is 140% volatile to the changes in the general stock market
rm = 0.086  # The stock in question will earn 8.6% as per historical data
# Calculate the expected return .
expected_return = expected_return(rf, beta, rm)
# Print the result.
print(expected_return)
```

Figure 12: An example of direct function call.

Problem & Planing

```
Read the following mathematical question and answer it. Please note that you could (optionally, not
required) call the following functions when writing the code:

Question:The linear density in a rod $8 \mathrm{~m}$ long is $12 / \sqrt{x+1} \mathrm{~kg} / \mathrm{m}$
, where $x$ is measured in meters from one end of the rod. Find the average density of the rod.
```

```
    (1) Integrate the linear density function with respect to x from 0 to 8.
    (2) Divide the result by the length of the rod.
```

Retrieved Functions

```python
def average_value_of_function(f, a, b):
    """
    Calculates the average value of a function `f` on the interval `[a, b]`.
    Parameters: - f (sympy expression): The function for which the average value is to be found.
    - a (float): The lower limit of the interval.
    - b (float): The upper limit of the interval.
    Returns: float: The average value of the function on the interval.
    """
    from sympy import integrate, symbols
    x = symbols('x')
    integral = integrate(f, (x, a, b))
    average_value = integral / (b - a)
    return average_value


def avg_distance(vectors):
    """
    Finds the average distance from the origin that can be reached by adding and subtracting the given
    vectors.
    Parameters: - vectors (list of numpy arrays): The vectors.
    Returns: float: The average distance.
    """
    from itertools import product
    import numpy as np
    total_distance = 0
    count = 0
    for signs in product([-1, 1], repeat=len(vectors)):
        resultant = sum(v * s for v, s in zip(vectors, signs))
        distance = np.linalg.norm(resultant)
        total_distance += distance
        count += 1
    return total_distance / count

...
```

Solution

```python
# The linear density is given by 12 / sqrt(x+1) kg/m.
# The average density is the total mass of the rod divided by its length.
# The total mass of the rod is the integral of the linear density times the length of the rod from 0 to
8.
from sympy import symbols, integrate
x = symbols('x')
length = 8
# Define the linear density function.
linear_density = 12 / (x + 1)**0.5
# Calculate the total mass of the rod.
total_mass = integrate(linear_density, (x, 0, length))
# Calculate the average density by dividing the total mass by the length.
average_density = total_mass / length
print(average_density)
```

Figure 13: An example of implicitly using functions as a reference.

H Prompts

H.1 Tool/Function Collection

Listing 5: Prompt used for planning and function collection

~~~
Given the question and the reference solution, do the following things:

- Think about what math knowledge points are required to solve this problem step by step.
- write some python one or more functions to abstract the solution. Please note that the functions should be well-documented as much as possible and not too specific (for example, do not write the values in this problem within the functions. Pass them as the function arguments). We hope your written functions could be re-used in anywhere else.
-Instantiate these functions to solve the problem. The last line of your program should be a 'print' command to print the final answer

Here are some examples you may refer to:

Question: There are integers $b,c$ for which both roots of the polynomial $x^2-x-1$ are also roots of the polynomial $x^5-bx-c$. Determine the product $bc$.
Answer: Let $r$ be a root of $x^2-x-1$. Then, rearranging, we have\n$$r^2 = r+1.$$Multiplying both sides by $r$ and substituting gives\n\\begin{align*}\nr^3 &= r^2+r \\\\\n&= (r+1)+r \\\\\n&= 2r+1.\n\\end{align*}Repeating this process twice more, we have\n\\begin{align*}\nr^4 &= r(2r+1) \\\\\n&= 2r^2+r \\\\\n&= 2(r+1)+r \\\\\n&= 3r+2\n\\end{align*}and\n\\begin{align*}\nr^5 &= r(3r+2) \\\\\n&= 3r^2+2r \\\\\n&= 3(r+1)+2r \\\\\n&= 5r+3.\n\\end{align*}Thus, each root of $x^2-x-1$ is also a root of $x^5-5x-3$, which gives $bc = 5\\cdot 3 = \\boxed{15}$.
Think: To solve this question, we can follow the steps below: (1) Find the roots of the polynomial $x^2-x-1$. (2) Substitute them into the the polynomial $x^5-bx-c$ and obtain two equations. (3) Solve the equations.
Functions:
```function 1
def find_roots_of_polynomial(polynomial, variable):
   """
   Finds the roots of a given polynomial using the sympy library.

   Parameters:
   - polynomial (sympy expression): The polynomial whose roots are to be found.
   - variable (sympy symbol): The variable of the polynomial.

   Returns:
   - list: The roots of the polynomial.
   """

   from sympy import solve
   roots = solve(polynomial, variable)
   return roots
```

```function 2
def substitute_roots_into_polynomial(roots, polynomial, variable):
   """
   Substitutes the given roots into the polynomial and returns the resulting expressions.

   Parameters:
   - roots (list): The list of roots to be substituted into the polynomial.
   - polynomial (sympy expression): The polynomial into which the roots are to be substituted.
   - variable (sympy symbol): The variable of the polynomial.

   Returns:
   - list: The resulting expressions after substituting the roots into the polynomial.
   """

   return [polynomial.subs(variable, root) for root in roots]
```
~~~

~~~
```function 3
def solve_equations(equations, variables):
   """
   Solves a system of equations for the specified variables using the sympy library.

   Parameters:
   - equations (list of sympy expressions or a single sympy expression):
      The equations to be solved. If solving a single equation, this can be a single
           expression.
   - variables (list of sympy symbols or a single sympy symbol):
      The variables for which the solution is to be found. If solving for a single
          variable, this can be a single symbol.

   Returns:
   - list of dictionaries: Each dictionary represents a solution, with keys being
       the variables and values being their corresponding values.
     If there's only one solution, the list will contain a single dictionary.
   """

   from sympy import solve
   solution = solve(equations, variables, dict=True)
   return solution
```

Solution:
```python
# Import required functions and classes from sympy
from sympy import symbols, Eq

# Define the variable and the polynomials
x, b, c = symbols('x b c')
polynomial1 = x**2 - x - 1
polynomial2 = x**5 - b*x - c

# Find the roots of the first polynomial
roots = find_roots_of_polynomial(polynomial1, x)

# Substitute the roots into the second polynomial
resulting_expressions = substitute_roots_into_polynomial(roots, polynomial2, x)

# Set up the equations based on the resulting expressions
equations = [Eq(expr, 0) for expr in resulting_expressions]

# Solve the system of equations for b and c
solutions = solve_equations(equations, (b, c))
# This linear system has only one solution
solution = solutions[0]

# Calculate the product bc
product_bc = solution[b] * solution[c]
print(product_bc)
```

---

Question: Medians $\\overline{DP}$ and $\\overline{EQ}$ of $\\triangle DEF$ are
    perpendicular. If $DP= 18$ and $EQ = 24$, then what is ${DE}$?
Answer: Point $G$ is the centroid of $\\triangle DEF$, so $DG:GP = EG:GQ = 2:1$.
    Therefore, $DG = \\frac23(DP) = 12$ and $EG = \\frac23(EQ) =16$, so applying the
     Pythagorean Theorem to $\\triangle EGD$ gives us $DE = \\sqrt{EG^2 + GD^2} = \\
    boxed{20}$.
Think: Given two perpendicular medians in a triangle, we need to perform the
    following steps: (1) Identify the relationship between the segments of medians
    and the centroid. (2) Use the ratios provided to determine the lengths of the
    individual segments from the centroid to the vertices. (3) Use the Pythagorean
    theorem to determine the length of the side connecting the two vertices from
    which the medians originate.
Functions:
```function 1
~~~

~~~
def median_segments_length(median_length, ratio):
   """
   Computes the lengths of the segments of a median split by the centroid.

   Parameters:
   - median_length (float): Total length of the median.
   - ratio (tuple): Ratio in which the centroid splits the median. Default is (2,1)
      for standard triangles.

   Returns:
   - tuple: Lengths of the two segments.

   Formula:
   - segment_1 = ratio[0]/sum(ratio) * median_length
   - segment_2 = ratio[1]/sum(ratio) * median_length
   """
   segment_1 = ratio[0] / sum(ratio) * median_length
   segment_2 = ratio[1] / sum(ratio) * median_length
   return segment_1, segment_2
```

```function 2
def pythagorean_theorem(a, b):
   """
   Computes the hypotenuse of a right triangle given two legs.

   Parameters:
   - a, b (float): Lengths of the two legs.

   Returns:
   - float: Length of the hypotenuse.

   Formula:
   - c = sqrt(a^2 + b^2)
   """
   from sympy import sqrt
   return sqrt(a**2 + b**2)
```

Solution:
```python
# Given values
DP = 18
EQ = 24

# Point $G$ is the centroid.
ratio = (2,1)
# Determine the lengths of the segments split by the centroid
DG, GP = median_segments_length(DP, ratio)
EG, GQ = median_segments_length(EQ, ratio)

# Use Pythagoras theorem to determine DE
DE = pythagorean_theorem(EG, DG)
print(DE)
```
~~~

H.2 Self-rectification

Listing 6: Prompt used for self-rectification

````
{
   "role": "system",
   "content": "Suppose you are writing Python programs to solve mathematical
      questions. Your original program, however, can not run normally for some
      reasons. Given the error message returned from the interpreter, please debug
      your program carefully and generate the new program. Your generated
      correction MUST FOLLOW the following format:\n```function 1\n[the first
      created function]\n```\n...\n```function N\n[the N-th created function]\n```\n
````

~~~
        ```python\n[solution]\n```.\nDO NOT only quote your output as ```python\n[
        solution]\n```"
    },
    {
        "role": "user",
        "content": "Original Program:\n```function 1\ndef solve_quadratic(a, b, c):\n
            from sympy import symbols, Eq, solve\n x = symbols('x')\n quadratic = Eq(a *
            x**2 + b * x + c, 0)\n solutions = solve(quadratic, x)\n return solutions\n
            ```\n\n```python\ndef find_d_values():\n a, b, c, d = symbols('a b c d')\n eq1
             = Eq(6 * a * d, a + d)\n eq2 = Eq((6 * d - 1) * (1 - a), a + d)\n eq3 = Eq(a,
             4 * d - 1)\n solution = solve([eq1, eq2, eq3], (a, b, c, d), dict=True)\n
            d_values = [sol[d] for sol in solution]\n return d_values\nd_values =
            find_d_values()\nprint(d_values)\n```\n\nError Message:\nFile <string>:15\n\
            nFile <string>:9, in find_d_values()\n\nNameError: name 'Eq' is not defined"
    },
    {
        "role": "assistant",
        "content": "Analysis: The error is occurring because the Eq function from the
            sympy module is not recognized in the find_d_values function. This is
            happening because the ```from sympy import symbols, Eq, solve``` line is only
            inside the ```solve_quadratic``` function, so Eq and other imported functions
            and classes are only available within the scope of that function. To fix this,
             you should claim import statement within the ```find_d_values``` function as
            well.\n\nNew Program:\n```function 1\ndef solve_quadratic(a, b, c):\n from
            sympy import symbols, Eq, solve\n x = symbols('x')\n quadratic = Eq(a * x**2
            + b * x + c, 0)\n solutions = solve(quadratic, x)\n return solutions\n```\n\n
            ```python\ndef find_d_values():\n from sympy import symbols, Eq, solve\n a, b,
             c, d = symbols('a b c d')\n eq1 = Eq(6 * a * d, a + d)\n eq2 = Eq((6 * d -
            1) * (1 - a), a + d)\n eq3 = Eq(a, 4 * d - 1)\n solution = solve([eq1, eq2,
            eq3], (a, b, c, d), dict=True)\n d_values = [sol[d] for sol in solution]\n
            return d_values\nd_values = find_d_values()\nprint(d_values)\n```"
    },
    {
        "role": "user",
        "content": "finished"
    },
    {
        "role": "user",
        "content": "Original Program:\n```function 1\ndef matrix_square(matrix):\n from
            sympy import Matrix\n if not isinstance(matrix, Matrix):\n matrix = Matrix(
            matrix)\n return matrix * matrix\n```\n\n```function 2\ndef solve_equations(
            equations, variables): \n from sympy import solve \n solution = solve(
            equations, variables, dict=True) \n return solution \n```\n\n```function 3\
            ndef count_solutions(solutions):\n return len(solutions)\n```\n\n```python\
            nfrom sympy import symbols, Eq \na, d = symbols('a d') \nmatrix = [[a, 3],
            [-8, d]] \nsquared_matrix = matrix_square(matrix) \nidentity_matrix = [[1, 0],
             [0, 1]] \nequations = [Eq(squared_matrix[i][j], identity_matrix[i][j]) for i
             in range(2) for j in range(2)] \nsolution = solve_equations(equations, [a, d
            ]) \nans = count_solutions(solution)\nprint(ans)\n```\n\nError Message:\nFile
             <string>:17\n\nFile <string>:17, in <listcomp>(.0)\n\nTypeError: 'Add'
            object is not subscriptable"
    },
    {
        "role": "assistant",
        "content": "Analysis: The error message is occurring because when you are
            creating the list of equations with the list comprehension, squared_matrix[i
            ][j] is not working as expected. The squared_matrix is a SymPy Matrix object,
             and you should use the .row and .col methods.\n\nNew Program:\n```function 1\
            ndef matrix_square(matrix):\n from sympy import Matrix\n if not isinstance(
            matrix, Matrix):\n matrix = Matrix(matrix)\n return matrix * matrix\n```\n\n
            ```function 2\ndef solve_equations(equations, variables): \n from sympy
            import solve \n solution = solve(equations, variables, dict=True) \n return
            solution \n```\n\n```function 3\ndef count_solutions(solutions):\n return len(
            solutions)\n```\n\n```python\nfrom sympy import symbols, Eq \na, d = symbols('
            a d') \nmatrix = [[a, 3], [-8, d]] \nsquared_matrix = matrix_square(matrix) \
            nidentity_matrix = [[1, 0], [0, 1]] \nequations = [Eq(squared_matrix.row(i)[j
            ], identity_matrix[i][j]) for i in range(2) for j in range(2)] \nsolution =
            solve_equations(equations, [a, d]) \nans = count_solutions(solution)\nprint(
            ans)\n```"
    },
~~~

```
{
   "role": "user",
   "content": "finished"
},
{
   "role": "user",
   "content": "Original Program:\n```function 1\ndef sum_to_product_cos(alpha, beta)
       :\n return (cmath.cos(alpha - beta) + cmath.cos(alpha + beta)) / 2\n```\n\n```
       function 2\ndef simplify_expression(expression): \n from sympy import
       simplify \n return simplify(expression) \n```\n\n```python\nfrom sympy import
       symbols, cos\n\nx = symbols('x')\nexpr = cos(x) + cos(3*x) + cos(7*x) + cos
       (9*x)\n\n# Apply sum-to-product formula\nexpr1 = 2 * sum_to_product_cos(5*x,
       4*x)\nexpr2 = 2 * sum_to_product_cos(5*x, 2*x)\n\n# Simplify the expression\
       nsimplified_expr = simplify_expression(expr1 + expr2)\n\n# Extract the values
        of a, b, c, and d and sum them up\na, b, c, d = 4, 1, 3, 5\nans = a + b + c
       + d\nprint(ans)\n```\n\nError Message:\nFile <string>:12\n\nFile <string>:2,
       in sum_to_product_cos(alpha, beta)\n\nNameError: name 'cmath' is not defined"
},
{
   "role": "assistant",
   "content": "Analysis: The error message is occurring because the cmath module is
       being used in the sum_to_product_cos function but it has not been imported.
       You can fix this by importing the cmath module in this function. New Program
       :\n```function 1\ndef sum_to_product_cos(alpha, beta):\n import cmath\n
       return (cmath.cos(alpha - beta) + cmath.cos(alpha + beta)) / 2\n```\n\n```
       function 2\ndef simplify_expression(expression): \n from sympy import
       simplify \n return simplify(expression)\n``` \n\n```python\nfrom sympy import
       symbols, cos\n\nx = symbols('x')\nexpr = cos(x) + cos(3*x) + cos(7*x) + cos
       (9*x)\n\n# Apply sum-to-product formula\nexpr1 = 2 * sum_to_product_cos(5*x,
       4*x)\nexpr2 = 2 * sum_to_product_cos(5*x, 2*x)\n\n# Simplify the expression\
       nsimplified_expr = simplify_expression(expr1 + expr2)\n\n# Extract the values
        of a, b, c, and d and sum them up\na, b, c, d = 4, 1, 3, 5\nans = a + b + c
       + d\nprint(ans)\n```"
},
{
   "role": "user",
   "content": "Error message:\nTraceback (most recent call last):\n File, line 9, in
        <module>\n exec(\n File '<string>', line 13, in <module>\n File '<string>',
       line 3, in sum_to_product_cos\n File, line 356, in __complex__\n return
       complex(float(re), float(im))\n File, line 351, in __float__\n raise
       TypeError('Cannot convert expression to float')\nTypeError: Cannot convert
       expression to float\n"
},
{
   "role": "assistant",
   "content": "Analysis: The error occurs because you are trying to use cmath.cos
       function (which expects float or complex numbers as arguments) with SymPy
       symbolic expressions (5x and 4x, 5x and 2x are symbolic expressions). To fix
       this, you should use SymPy's cos function instead of cmath.cos function to
       work with symbolic expressions.\n\nNew Program:\ndef sum_to_product_cos(alpha,
        beta):\n from sympy import cos\n return (cos(alpha - beta) + cos(alpha +
       beta)) / 2\ndef simplify_expression(expression): \n from sympy import
       simplify \n return simplify(expression) \nfrom sympy import symbols, cos\n\nx
        = symbols('x')\nexpr = cos(x) + cos(3*x) + cos(7*x) + cos(9*x)\n\n# Apply
       sum-to-product formula\nexpr1 = 2 * sum_to_product_cos(5*x, 4*x)\nexpr2 = 2 *
        sum_to_product_cos(5*x, 2*x)\n\n# Simplify the expression\nsimplified_expr =
        simplify_expression(expr1 + expr2)\n\n# Extract the values of a, b, c, and d
        and sum them up\na, b, c, d = 4, 1, 3, 5\nans = a + b + c + d\nprint(ans)"
},
{
   "role": "user",
   "content": "finished"
},
{
   "role": "user",
   "content": "Original Program:\n```function 1\ndef tan_double_angle(tan_theta):\n
       return 2 * tan_theta / (1 - tan_theta**2)\n```\n\n```function 2\ndef
       tan_triple_angle(tan_theta):\n return (3 * tan_theta - tan_theta**3) / (1 - 3
        * tan_theta**2)\n```\n\n```function 3\ndef solve_polynomial(polynomial, x):\n
        from sympy import solve\n return solve(polynomial, x)\n```\n\n```python\nfrom
```

~~~
         sympy import symbols\ntan_theta = symbols('tan_theta')\ntan_2_theta =
        tan_double_angle(tan_theta)\ntan_3_theta = tan_triple_angle(tan_theta)\
        nequation = tan_theta + tan_2_theta + tan_3_theta\nsimplified_equation =
        simplify_expression(equation)\ntan_theta_solutions = solve_polynomial(
        simplified_equation, tan_theta)\n\n# Find the solution in the given range (0
        < tan_theta < 1)\ntan_theta_value = [solution for solution in
        tan_theta_solutions if 0 < solution < 1][0]\nprint(tan_theta_value)\n```\n\
        nError Message:\nFile <string>:13\n\nNameError: name 'simplify_expression' is
         not defined"
},
{
   "role": "assistant",
   "content": "Analysis: The error message is occurring because there is no function
         named simplify_expression defined in your script. To fix this, you can
        define such a function that uses SymPy's simplify function to simplify
        expressions.\n\nNew Program:\n```function 1\ndef simplify_expression(
        expression): \n from sympy import simplify \n return simplify(expression)\n
        ```\n\n```function 2\ndef tan_double_angle(tan_theta):\n return 2 * tan_theta
        / (1 - tan_theta**2)\n```\n\n```function 3\ndef tan_triple_angle(tan_theta):\n
         return (3 * tan_theta - tan_theta**3) / (1 - 3 * tan_theta**2)\n```\n\n```
        function 4\ndef solve_polynomial(polynomial, x):\n from sympy import solve\n
        return solve(polynomial, x)\n```\n\n```python\nfrom sympy import symbols\
        ntan_theta = symbols('tan_theta')\ntan_2_theta = tan_double_angle(tan_theta)\
        ntan_3_theta = tan_triple_angle(tan_theta)\nequation = tan_theta +
        tan_2_theta + tan_3_theta\nsimplified_equation = simplify_expression(equation
        )\ntan_theta_solutions = solve_polynomial(simplified_equation, tan_theta)\n\n
        # Find the solution in the given range (0 < tan_theta < 1)\ntan_theta_value =
         [solution for solution in tan_theta_solutions if 0 < solution < 1][0]\nprint
        (tan_theta_value)\n```"
},
{
   "role": "user",
   "content": "finished"
}
~~~

H.3 Function-augmented Solutions

Listing 7: Prompt used for the generation of function-augmented solutions (cross-retrieval strategy)

~~~
You will encounter a mathematical problem and are required to write a piece of
    Python code to solve this problem.

Now we have a suite of wrapped functions. Take note:
- The newly provided wrapped functions have NOT been verified. They may be
    irrelevant or potentially flawed.
- It's essential that the solution doesn't overly depend on wrapped functions.
  You're welcome to utilize one or more functions from the new set in your solution
      but only after you've determined:
  (1) Their accuracy.
  (2) Their inclusion significantly streamlines the problem-solving approach.

Additionally take note that
   (1) The last line of your written code shall be a 'print' command to print the
       final answer.
   (2) The wrapped functions should not be duplicated within your code. Instead,
       call them directly if needed.
   (3) Should you need to create custom functions, do so without adding
       documentation comments for the sake of brevity.
   (4) Write simple but clear annotations interleaving your code solution.

"""
Retrieved functions:
[List of called function names from the new set]

```python
[Your Written Python Code.]
```
"""
~~~

~~~
For example:
---
Question: What is the 100th digit to the right of the decimal point in the decimal
    representation of $\frac{13}{90}$?

New provided functions:
```New Function 0
def decimal_representation(numerator, denominator, max_digits=1000):
   """
   Computes the decimal representation of a fraction.

   Parameters:
   - numerator (int): The numerator of the fraction.
   - denominator (int): The denominator of the fraction.
   - max_digits (int): The maximum number of decimal digits to compute.

   Returns:
   - str: The decimal representation of the fraction as a string.
   """

   result = ""
   remainder = numerator % denominator
   for _ in range(max_digits):
      remainder *= 10
      result += str(remainder // denominator)
      remainder %= denominator
      if remainder == 0:
         break
   return result
```

```New Function 1
def decimal_to_scientific(decimal_number):
   from sympy import log, floor
   exponent = -floor(log(decimal_number, 10))
   coefficient = decimal_number * 10**(-exponent)
   return coefficient, exponent
```

```New Function 2
def repeating_decimal_representation(numerator, denominator):
   """
   Computes the repeating decimal representation of a fraction.

   Parameters:
   - numerator (int): The numerator of the fraction.
   - denominator (int): The denominator of the fraction.

   Returns:
   - str: The repeating decimal representation of the fraction as a string.
   """

   # Initialize the result string and a dictionary to store remainders.
   result = ""
   remainders = {}

   # Perform long division to find the decimal representation.
   while numerator != 0:
      # If the remainder has been seen before, we found the repeating block.
      if numerator in remainders:
         start = remainders[numerator]
         return result[:start] + "(" + result[start:] + ")"
      # Otherwise, store the remainder and continue the division.
      remainders[numerator] = len(result)
      numerator *= 10
      result += str(numerator // denominator)
      numerator %= denominator

   return result
```
~~~

~~~
```New Function 3
def nth_digit_of_decimal_representation(numerator, denominator, n):
   """
   Computes the nth digit after the decimal point of the decimal representation of a
        fraction.

   Parameters:
   - numerator (int): The numerator of the fraction.
   - denominator (int): The denominator of the fraction.
   - n (int): The position of the digit after the decimal point.

   Returns:
   - int: The nth digit after the decimal point of the decimal representation of the
        fraction.
   """

   # Get the repeating decimal representation of the fraction.
   decimal_representation = repeating_decimal_representation(numerator, denominator)

   # Remove the parentheses from the repeating block.
   decimal_representation = decimal_representation.replace("(", "").replace(")", "")

   # Calculate the nth digit using the repeating block.
   return int(decimal_representation[(n - 1) % len(decimal_representation)])
```

Retrieved functions:
[decimal_representation, nth_digit_of_decimal_representation]

```python
# Use the nth_digit_of_decimal_representation function to find the 100th digit
numerator = 13
denominator = 90
n = 100

# Call the function and print the result
result = nth_digit_of_decimal_representation(numerator, denominator, n)
print(result)
```



---
Question: The square root of $x$ is greater than 3 and less than 4. How many integer
     values of $x$ satisfy this condition?

New provided functions:
```New Function 0
def solve_square_root_equation(a, b, c):
   """
   Solves a square root equation of the form sqrt(ax - b) = c.

   Parameters:
   - a (float): Coefficient of x inside the square root.
   - b (float): Constant term inside the square root.
   - c (float): Constant term on the right side of the equation.

   Returns:
   - float: The value of x that satisfies the equation.

   Formula:
   - x = (c^2 + b) / a
   """
   return (c**2 + b) / a
```

```New Function 1
def find_integer_square_less_than_double():
   """
   Finds the only integer whose square is less than its double.
~~~

~~~
   Returns:
   - int: The integer that satisfies the condition.

   Method:
   - Iterate through integers starting from 1, and check if the square of the
       integer is less than its double.
   - If the condition is satisfied, return the integer.
   - If the condition is not satisfied for any integer up to a certain limit, return
        None.
   """
   limit = 100
   for x in range(1, limit):
      if x**2 < 2*x:
         return x
   return None
```

```New Function 2
def solve_equation():
   """
   Solves the equation (x-2)^(25-x^2) = 1 for integer solutions.

   Returns:
   - list: A list of integer solutions for x.
   """
   solutions = []

   # Case 1: Exponent is 0 (25 - x^2 = 0)
   x1 = 5
   x2 = -5
   solutions.extend([x1, x2])

   # Case 2: Base is 1 (x - 2 = 1)
   x3 = 3
   solutions.append(x3)

   # Case 3: Base is -1 and exponent is even (x - 2 = -1 and 25 - x^2 = 2n for some
       integer n)
   x4 = 1
   solutions.append(x4)

   return solutions
```

```New Function 3
def count_integers_in_range(lower_bound, upper_bound, exclude_zero=True):
   """
   Counts the number of integers within a given range.

   Parameters:
   - lower_bound (int): The lower bound of the range.
   - upper_bound (int): The upper bound of the range.
   - exclude_zero (bool): Whether to exclude 0 from the count. Default is True.

   Returns:
   - int: The number of integers within the range.
   """
   count = upper_bound - lower_bound + 1
   if exclude_zero and lower_bound <= 0 and upper_bound >= 0:
      count -= 1
   return count
```

Retrieved functions:
[]

```python
# The lower and upper bounds of x for which sqrt(x) > 3 and sqrt(x) < 4
lower_bound = 9
~~~

~~~
upper_bound = 16

# Counting the number of integers between 9 (exclusive) and 16 (exclusive)
num_integers = len([x for x in range(lower_bound + 1, upper_bound)])

# Printing the result
print(num_integers)
```
~~~

H.4 Evaluation with Toolsets

Listing 8: Prompt used for evaluation (setting with toolsets)

```
Read the following questions and answer them. For each question, you are required to
     write a Python program to solve it.
Please note that we provide you several functions for each question. You could (
    optionally, not required) call the functions to help you to solve the question
    if necessary.
Note that the last line of your program should be a 'print' command to print the
    final answer

----------------------------------------------------
Question:
What is the 100th digit to the right of the decimal point in the decimal
    representation of $\\frac{13}{90}$?

Functions:
def repeating_decimal_representation(numerator, denominator):
   """
   Computes the repeating decimal representation of a fraction.

   Parameters:
   - numerator (int): The numerator of the fraction.
   - denominator (int): The denominator of the fraction.

   Returns:
   - str: The repeating decimal representation of the fraction as a string.
   """

   # Initialize the result string and a dictionary to store remainders.
   result = ""
   remainders = {}

   # Perform long division to find the decimal representation.
   while numerator != 0:
      # If the remainder has been seen before, we found the repeating block.
      if numerator in remainders:
         start = remainders[numerator]
         return result[:start] + "(" + result[start:] + ")"
      # Otherwise, store the remainder and continue the division.
      remainders[numerator] = len(result)
      numerator *= 10
      result += str(numerator // denominator)
      numerator %= denominator

   return result


def nth_digit_of_decimal_representation(numerator, denominator, n):
   """
   Computes the nth digit after the decimal point of the decimal representation of a
        fraction.

   Parameters:
   - numerator (int): The numerator of the fraction.
   - denominator (int): The denominator of the fraction.
   - n (int): The position of the digit after the decimal point.

   Returns:
```

```
   - int: The nth digit after the decimal point of the decimal representation of the
        fraction.
   """

   # Get the repeating decimal representation of the fraction.
   decimal_representation = repeating_decimal_representation(numerator, denominator)

   # Remove the parentheses from the repeating block.
   decimal_representation = decimal_representation.replace("(", "").replace(")", "")

   # Calculate the nth digit using the repeating block.
   return int(decimal_representation[(n - 1) % len(decimal_representation)])


def decimal_representation(numerator, denominator, max_digits=1000):
   """
   Computes the decimal representation of a fraction.

   Parameters:
   - numerator (int): The numerator of the fraction.
   - denominator (int): The denominator of the fraction.
   - max_digits (int): The maximum number of decimal digits to compute.

   Returns:
   - str: The decimal representation of the fraction as a string.
   """

   result = ""
   remainder = numerator % denominator
   for _ in range(max_digits):
      remainder *= 10
      result += str(remainder // denominator)
      remainder %= denominator
      if remainder == 0:
         break
   return result


Solution:
# find the 100th digit.
numerator = 13
denominator = 90
n = 100

# Call the function and print the result.
result = nth_digit_of_decimal_representation(numerator, denominator, n)
print(result)


----------------------------------------------------
Question:
The square root of $x$ is greater than 3 and less than 4. How many integer values of
     $x$ satisfy this condition?

Functions:
def count_integers_in_range(lower_bound, upper_bound, exclude_zero=True):
   """
   Counts the number of integers within a given range.

   Parameters:
   - lower_bound (int): The lower bound of the range.
   - upper_bound (int): The upper bound of the range.
   - exclude_zero (bool): Whether to exclude 0 from the count. Default is True.

   Returns:
   - int: The number of integers within the range.
   """
   count = upper_bound - lower_bound + 1
   if exclude_zero and lower_bound <= 0 and upper_bound >= 0:
      count -= 1
```

```
    return count


def find_integer_square_less_than_double():
    """
    Finds the only integer whose square is less than its double.

    Returns:
    - int: The integer that satisfies the condition.

    Method:
    - Iterate through integers starting from 1, and check if the square of the
       integer is less than its double.
    - If the condition is satisfied, return the integer.
    - If the condition is not satisfied for any integer up to a certain limit, return
        None.
    """
    limit = 100
    for x in range(1, limit):
        if x**2 < 2*x:
            return x
    return None


def solve_square_root_equation(a, b, c):
    """
    Solves a square root equation of the form sqrt(ax - b) = c.

    Parameters:
    - a (float): Coefficient of x inside the square root.
    - b (float): Constant term inside the square root.
    - c (float): Constant term on the right side of the equation.

    Returns:
    - float: The value of x that satisfies the equation.

    Formula:
    - x = (c^2 + b) / a
    """
    return (c**2 + b) / a


Solution:
# We need to find the integer values of x for which sqrt(x) > 3 and sqrt(x) < 4. To
    this end, we can count the number of integers in the range (9, 16) directly.
result = 0
for x in range(10, 16):
    if 9 < x < 16:
        result += 1
print(result)
```

H.5 Evaluation without Toolsets

Listing 9: Prompt used for evaluation (setting without toolsets)

```
Read the following questions and answer them. For each question, you are required to
     write a Python program to solve it.
Please note that we provide you several functions for each question. You could (
    optionally, not required) call the functions to help you to solve the question
    if necessary.
Note that the last line of your program should be a 'print' command to print the
    final answer

----------------------------------------------------
Question:
What is the 100th digit to the right of the decimal point in the decimal
    representation of $\\frac{13}{90}$?

Solution:
```

```
from decimal import Decimal, getcontext

# Set the precision to 101 (100 digits after decimal + 1 digit before decimal)
getcontext().prec = 101

# Calculate the decimal representation of 13/90
dec = Decimal(13) / Decimal(90)

# Convert the decimal to a string
dec_str = str(dec)

# Get the 100th digit to the right of the decimal point
digit_100th = dec_str[101]

print(digit_100th)

----------------------------------------------------
Question:
The square root of $x$ is greater than 3 and less than 4. How many integer values of
     $x$ satisfy this condition?

Solution:
# We need to find the integer values of x for which sqrt(x) > 3 and sqrt(x) < 4. To
    this end, we can count the number of integers in the range (9, 16) directly.
result = 0
for x in range(10, 16):
   if 9 < x < 16:
      result += 1
print(result)
```