

PRiSM: An Agentic Multimodal Benchmark for Scientific Reasoning via Python-Grounded Evaluation

Shima Imani, Seungwhan Moon, Adel Ahmadyan, Lu Zhang, Kirmani Ahmed, Babak Damavandi

Meta Reality Lab

Evaluating vision-language models (VLMs) in scientific domains like mathematics and physics poses unique challenges that go far beyond predicting final answers. These domains demand conceptual understanding, symbolic reasoning, and adherence to formal laws, requirements that most existing benchmarks fail to address. In particular, current datasets tend to be static, lacking intermediate reasoning steps, robustness to variations, or mechanisms for verifying scientific correctness. To address these limitations, we introduce PRiSM, a synthetic, fully dynamic, and multimodal benchmark for evaluating scientific reasoning via grounded Python code. PRiSM includes over 24,750 university-level physics and math problems, and it leverages our scalable agent-based pipeline, PrismAgent, to generate well-structured problem instances. Each problem contains dynamic textual and visual input, a generated figure, alongside rich structured outputs: executable Python code for ground truth generation and verification, and detailed step-by-step reasoning. The dynamic nature and Python-powered automated ground truth generation of our benchmark allow for fine-grained experimental auditing of multimodal VLMs, revealing failure modes, uncertainty behaviors, and limitations in scientific reasoning. To this end, we propose five targeted evaluation tasks covering generalization, symbolic program synthesis, perturbation robustness, reasoning correction, and ambiguity resolution. Through comprehensive evaluation of existing VLMs, we highlight their limitations and showcase how PRiSM enables deeper insights into their scientific reasoning capabilities.

Date: December 8, 2025

Correspondence: First Author at shimaimani@meta.com



1 Introduction

Effective reasoning is fundamental for systematic problem-solving, logical deduction, and structured decision-making. In complex scientific domains like mathematics and physics, accurate reasoning requires the explicit integration of theoretical principles, rigorous mathematical processes, and computational verification to ensure dimensional and numerical validity [Lake et al. \(2017\)](#); [Polya \(2014\)](#); [Chi et al. \(1981\)](#); [Newell and Simon \(1972\)](#).

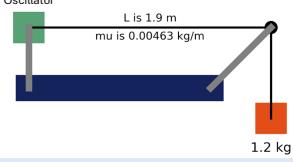
Recent advancements in multimodal VLMs have significantly improved their reasoning capabilities. Innovations like advanced prompting techniques (e.g., chain-of-thought, tree-of-thought, and self-reflection), supervised fine-tuning on reasoning tasks, direct preference optimization (DPO), and reinforcement learning with human feedback (RLHF) have contributed to these improvements [Kojima et al. \(2022\)](#); [Wei et al. \(2022\)](#); [Yao et al. \(2023\)](#); [Renze and Guven \(2024\)](#); [Ouyang et al. \(2022\)](#); [Rafailov et al. \(2023\)](#). Alongside these advancements, a variety of datasets and benchmarks have emerged, explicitly designed to assess and enhance the reasoning skills of models in scientific contexts [Lu et al. \(2022\)](#); [Wang et al. \(2023\)](#); [Hendrycks et al. \(2021\)](#); [Sun et al. \(2024\)](#).

Nonetheless, existing benchmarks remain limited in several critical ways:

- **Limited Generalization:** Most benchmarks rely on static problem formulations, lacking systematic variations or paraphrases. Furthermore, the visual modality is typically static. Consequently, these datasets fail to rigorously assess models' generalization and robustness across diverse problem settings and parameter variations.
- **Missing Intermediate Reasoning:** The majority of datasets provide only final numerical answers, omitting detailed intermediate reasoning steps. This absence of granular guidance hinders a comprehensive evaluation of a

Question

Consider a rope of length L attached to a variable-frequency oscillator. The rope has a linear mass density of μ and is suspended over a frictionless, massless pulley with a weight of m attached to its end, providing tension. (a) Determine the speed of the waves propagating through the rope. (b) Calculate the first three resonant modes of the standing waves that can form on the rope, labeling each with its corresponding wavelength. (c) Identify the frequencies to which the oscillator must be tuned to produce these first three resonant modes. Here are the constants: $g=9.8\text{m/s}^2$



Reasoning Steps

Identify Relevant Concepts:

- The velocity of a wave on a string is determined by the tension and the linear mass density of the string. The standing waves on the string are subject to boundary conditions, specifically that there must be a node at each end of the string. The target variables are the velocity of the wave, the wavelength of the first three normal modes, and the frequencies required to produce those modes.
- Known quantities include the length of the string 1.9 m, the linear mass density 0.00463 kg/m, and the mass of the hanging weight 1.2 kg.

Set Up the Problem:

- The velocity of the wave can be found using the formula $v = \sqrt{F_T / \mu}$, where F_T is the tension provided by the hanging mass ($F_T = m \cdot g$).
- For standing waves, the boundary conditions dictate that there must be a node at each end, meaning the first mode will be half a wavelength, and subsequent modes are found by adding half wavelengths.
- The relationship between wave speed, wavelength, and frequency is given by $v = \lambda f$, which can be rearranged to find frequency ($f = v / \lambda$).

Execute the Solution:

- Calculate the velocity of the wave using the given tension and linear mass density: $v = \sqrt{\mu \cdot g}$.
- Determine the wavelengths of the first three normal modes based on the boundary conditions and the length of the string. The first mode is half a wavelength, the second mode is one wavelength, and the third mode is one and a half wavelengths.
- Calculate the frequencies of the first three normal modes using the formula $f = v / \lambda$, where λ is the wavelength of each mode.

Evaluate Your Answer:

- The solution for the wave velocity and the frequencies of the normal modes should be reasonable and consistent with the physical principles of wave propagation on a string.
- Verify that the calculated frequencies would produce standing waves with nodes at the ends of the string, as required by the boundary conditions.

Python Code Synthesis

```

import sympy as sp
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity

def wave_on_string(L, mu, m, g):
    # Convert inputs to Pint quantities
    L = Q_(L).to(ureg.meter) # Ensure meters
    mu = Q_(mu).to(ureg.kilogram / ureg.meter) # Ensure kg/m
    m = Q_(m).to(ureg.kilogram) # Ensure kg
    g = Q_(g).to(ureg.meter / ureg.second**2) # Ensure m/s

    # Strip units for symbolic computation
    L = L.magnitude
    mu = mu.magnitude
    m = m.magnitude
    g = g.magnitude

    # (a) Calculate the velocity of the wave on the string
    v = sp.sqrt(m * g / mu)

    # (b) Calculate the wavelengths for the first three normal modes
    lambda_1 = 2 * L
    lambda_2 = L
    lambda_3 = 2 * L / 3

    # (c) Calculate the frequencies for the first three normal modes
    f_1 = v / lambda_1
    f_2 = v / lambda_2
    f_3 = v / lambda_3

    # Convert the results to numerical values
    v = v.evalf()
    f_1 = f_1.evalf()
    f_2 = f_2.evalf()
    f_3 = f_3.evalf()

    return {
        'wave_velocity': v,
        'wavelength_1': lambda_1,
        'wavelength_2': lambda_2,
        'wavelength_3': lambda_3,
        'frequency_1': f_1,
        'frequency_2': f_2,
        'frequency_3': f_3
    }

```

Figure 1 An instance from the PRiSM dataset includes a parameterized question with substituted input variables, a figure generated with problem inputs, a step-by-step solution, and the corresponding Python code.

model's reasoning path transparency, logical coherence, and interpretability.

- Lack of Integrated Computational Verification:** Existing benchmarks generally do not offer computational mechanisms to rigorously evaluate the quantitative correctness and dimensional consistency of solutions, crucial for scientific reasoning in mathematics and physics.
- Lack of Fine-Grained Error Analysis:** Existing benchmarks often focus on overall accuracy, without providing detailed breakdowns of the specific types of errors models make. This lack of detailed error categorization limits the understanding of model weaknesses and hinders targeted improvement.

To address these limitations, we present **PRiSM** (**P**ython-based **R**easoning for **S**cience and **M**ath), a fully dynamic, synthetic, and multimodal benchmark that systematically targets the identified gaps. Using our scalable agent-based pipeline, PrismAgent, we generate over 24,750 university-level mathematics and physics problem instances that cover a broad spectrum of scientific reasoning tasks. Its dynamic nature allows us to define a suite of diagnostic tasks targeting diverse dimensions of reasoning beyond final correctness. Our core contributions are as follows:

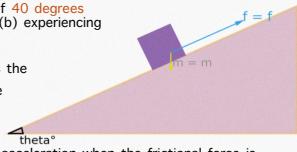
- An agentic data generation pipeline (PrismAgent):** A scalable pipeline that leverages autonomous agents to generate diverse scientific problems. Its modular design allows for efficient expansion to new domains and task types.
- A dynamic, multimodal dataset:** Each problem instance includes a parameterized textual prompt and programmatically generated figure, both tied to the same variable inputs. Paraphrased versions of the text are also included to assess linguistic robustness.
- Structured step-by-step solutions:** We provide symbolic derivations and detailed reasoning steps, allowing fine-grained evaluation of logical coherence and intermediate understanding.
- Executable Python code:** Each instance includes code that computes the ground truth solution using libraries such as SymPy (for symbolic math) and Pint (for unit consistency), supporting automated verification.
- Five diagnostic benchmark tasks:** Each task is designed to probe different reasoning capabilities, including generalization, symbolic program synthesis, robustness to visual perturbation, correction of flawed reasoning, and

Task I: Robustness to Input Variations: Input variables and wording vary across instances of the same problem.

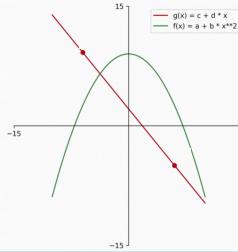
I) Imagine an object of mass **70.3 kg** sliding down a frictionless inclined plane that makes an angle of **40 degrees** with the horizontal. What would be its acceleration if it were (a) sliding without any resistance, and (b) experiencing a frictional force of **46.8 N**?

II) A cyclist with a mass of **71.4 kg** is riding down a hill inclined at an angle **35 degrees**. (a) What is the acceleration of the cyclist if there is no air resistance or friction? (b) What is the acceleration of the cyclist if there is a frictional force of **29.7 N** acting against the motion?

III) A snowboarder with a mass of **64.0 kg** is gliding down a slope of **63 degrees**. (a) Determine her acceleration when the frictional force is negligible. (b) Calculate her acceleration when the frictional force is **34.5 N**.



Task III: Incorrect Reasoning: Different types of partially incorrect solutions are presented.



Consider a region bounded by two curves, one represented by the function $f(x) = 6.6 + -0.197x^2$ and the other by $g(x) = 3.4 + -0.68x$. Determine the area enclosed between these two curves from $(-6, 6)$.

Unfamiliarity Error: I'll just plug in some x values into $f(x)$ and $g(x)$ and see what happens. For $x = -6$, $f(-6) = 9 + (-0.179)(-6)^2 = 2.556$ and $g(-6) = 9.18$. For $x = 6$, $f(6) = 2.556$ again and $g(6) = -4.98$. So, the area is like, the difference between these values, right?

Encoding Error: To find the area, I need to integrate the sum of the two functions over the interval where they intersect. So, I will calculate $\int(f(x)+g(x)) dx$ from $x = -6$ to $x = 6$.

Figure 2 Examples from the PRiSM dataset illustrating Task I (Robustness to Input Variations) and Task III (Reasoning with Correction). For Task I, we vary input values and paraphrase the problem text to evaluate numerical generalization. For Task III, we introduce different types of incorrect reasoning steps and assess the model’s ability to detect and correct mistakes in multi-step solutions.

reasoning under ambiguity. Figure 2 illustrates representative examples from some of these tasks.

2 Related Work

| Dataset | Size | Level | Reasoning Steps | Code Synthesis | Textual Variation (Q&A) | Numerical Variation (Q&A) | Dynamic Generated Figure |
|---------------------------------|---------------|--------------------|-----------------|----------------|-------------------------|---------------------------|--------------------------|
| ScienceQA Lu et al. (2022) | 21,208 | Elem. & Highschool | ✗ | ✗ | ✗ | ✗ | ✗ |
| SciBench Wang et al. (2023) | 594 | Highschool | ✗ | ✗ | ✗ | ✗ | ✗ |
| SciEval Sun et al. (2024) | 1,657 | Mixed | ✗ | ✗ | ✗ | ✗ | ✗ |
| JEEBench Arora et al. (2023) | 512 | Highschool/College | ✗ | ✗ | ✗ | ✗ | ✗ |
| MMMU Yue et al. (2024) | 11,500 | College | Partial | ✗ | ✗ | ✗ | ✗ |
| PRiSM (Ours) | 24,750 | College | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 1 Comparison of scientific reasoning datasets. PRiSM provides dynamic question variations (textual and numerical), structured reasoning, executable code, and generated figures capabilities.

Scientific Reasoning and VLMs. Vision-language models (VLMs) have demonstrated strong capabilities in general-purpose multimodal reasoning, aided by methods such as chain-of-thought prompting Wei et al. (2022), tree-of-thought Yao et al. (2023), self-reflection Renze and Guven (2024), instruction tuning Ouyang et al. (2022), and reinforcement learning with human feedback (RLHF) Christiano et al. (2017). While these techniques significantly enhance model reasoning across many tasks, scientific domains such as mathematics and physics remain especially challenging. Solving problems in these areas often requires precise symbolic manipulation, dimensional analysis, and adherence to formal physical laws, skills that are not explicitly targeted by most existing training paradigms or benchmarks Hendrycks et al. (2021); Lake et al. (2017); Chi et al. (1981).

Benchmarks for Scientific Reasoning. Several datasets have been introduced to evaluate scientific reasoning. ScienceQA Lu et al. (2022) provides multimodal science questions aimed at grade school curricula but focuses only on final

answers. SciBench Wang et al. (2023) covers multiple scientific domains using text but lacks step-by-step reasoning or executable validation. MMMU Yue et al. (2024) includes college-level multimodal questions but does not offer symbolic solutions or visual perturbations. JEEBench Arora et al. (2023) and SciEval Sun et al. (2024) target standardized exams but provide minimal reasoning supervision and no programmatic verification. Overall, these benchmarks are mostly static and do not support fine-grained auditing, dynamic variation, or computational verification.

Executable and Tool-Augmented Reasoning. Recent work has shown that integrating external tools improves model performance on reasoning tasks. Program-aided reasoning frameworks such as PAL Gao et al. (2023), Toolformer Schick et al. (2023), and plugin-based environments OpenAI (2023) allow models to use symbolic computation and external APIs to generate and verify answers. These frameworks highlight the value of structured program synthesis and execution for rigorous validation, an approach our benchmark adopts at scale through Python-based symbolic reasoning and verification.

Synthetic Benchmarks and Model Auditing. Synthetic data offers a controlled, scalable means to study model behavior, generalization, and failure modes Ding et al. (2021); Wang et al. (2024). Dynamic synthetic benchmarks can isolate reasoning abilities and test robustness by introducing input perturbations or ambiguous cases. Despite this potential, most existing science-focused benchmarks remain static and cannot probe how VLMs respond to controlled variations. Our work builds on this motivation by offering a fully dynamic benchmark with executable ground truth, enabling fine-grained audits of multimodal reasoning in scientific domains. Additionally, we define five benchmark tasks for a comprehensive evaluation of existing VLMs.

Table 1 provides a detailed comparison, highlighting the distinctive features and contributions of PRISM in relation to existing benchmarks.

3 PRISM: Benchmark Overview

3.1 Methodology

In this section, we describe the construction of the agent-based pipeline PrismAgent and the structure of the resulting dataset. Figure 3 provides an overview of our automated pipeline PrismAgent for constructing the PRISM benchmark.

Importantly, all final examples were manually reviewed and filtered to include only correct and consistent instances, ensuring alignment between the question, code, and generated figure.

3.2 OCR Extraction

We begin by converting undergraduate-level physics and mathematics materials (primarily in PDF format) into images. All original problem sources in PRISM are derived from openly available Creative Commons licensed content under the CC-BY-NC license¹. Using Optical Character Recognition (OCR) in combination with heuristic layout analysis, we segment and crop regions containing problem statements, solutions, and relevant figures. These regions serve as inputs to our automated benchmark generation pipeline.

3.3 PrismAgent: Agent for Structured Extraction

Image-to-Text. To convert cropped regions into structured textual data, PrismAgent leverages a vision-language model (e.g., LLaMa-3.2-90B-Vision-Instruct Grattafiori et al. (2024)) to extract the full *Question* and associated *Reasoning Steps*. This preserves mathematical notation, symbols, and formatting crucial for accurate representation and downstream computation.

PrismAgent also generates an *Input Map*, capturing all numerical variables and physical constants (e.g., {"a": [2, "dimensionless"]}), and an *Output Map*, containing the expected final results with corresponding units (e.g., {"v_a_final": [-3, "m/s"]}). These structured maps enable parameterized code generation and scalable variation.

¹Portions of the data were generated by LLaMA versions 3.2 and 3.3 and are subject to their respective licenses: https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/LICENSE and https://github.com/meta-llama/llama-models/blob/main/models/llama3_3/LICENSE.

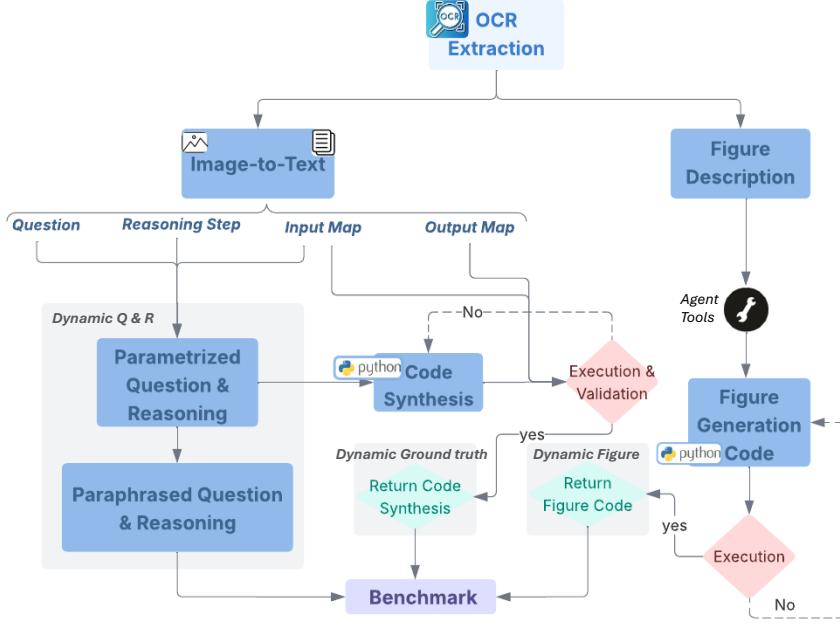


Figure 3 Overview of the Dataset Creation Pipeline Enabled by Our Framework.

Figure Description. PrismAgent produces a step-by-step description of the extracted figure by annotating only the essential elements required to solve the problem. These annotations are abstracted to enable reconstruction of the figure using Python code, avoiding reliance on the original visual appearance and enabling dynamic figure generation.

Parameterized Question & Reasoning. To enable abstraction and generalization, PrismAgent replaces concrete values in the question and reasoning steps with placeholders derived from the *Input Map* and *Output Map*. This produces a standardized, symbolic representation that facilitates controlled variation and code synthesis.

Paraphrased Question & Reasoning. To assess model robustness and linguistic generalization, PrismAgent generates paraphrased versions of the parameterized question and reasoning steps. These paraphrases retain the original logical structure and variable placeholders while varying the phrasing, context, and sentence structure. This allows for the creation of diverse yet semantically equivalent instances. Although these dynamic variations lack direct executable ground truth, this is addressed in the following step via code generation from the paraphrased reasoning.

Code Synthesis. Building on the symbolic reasoning steps and structured metadata, PrismAgent synthesizes executable Python functions that fully capture the solution logic for each problem instance. Each function accepts symbolic keys from the *Input Map* as inputs and returns results mapped to the corresponding keys in the *Output Map*.

To guarantee scientific validity, all code relies on established libraries: *Pint* enforces unit consistency throughout computations, while *Sympy* supports symbolic manipulation, algebraic simplification, and equation solving [Pint Developers \(2025\)](#); [Meurer et al. \(2017\)](#).

This step completes the dynamic solution pipeline by enabling the generation of ground-truth answers: substituting concrete numerical values into the inputs produces the correct output for any paraphrased or parameterized instance.

Execution and Validation. To ensure scientific correctness and computational reliability, each synthesized Python function undergoes a two-stage validation process. First, dimensional validation verifies that all physical quantities adhere to correct unit consistency using the *Pint* library. This step is primarily relevant for physics problems involving units, and it guarantees that operations are dimensionally coherent. For problems without units, this validation trivially passes.

Second, numerical validation executes the generated function by substituting concrete values from the *Input Map*, then compares the outputs against the expected numerical results specified in the *Output Map*. This step verifies that the

Python code produces the correct output for the original input values of the problem. If either check fails, PrismAgent automatically flags the failure and performs iterative correction until both validations pass.

Figure Generation. To reconstruct problem diagrams, PrismAgent uses symbolic figure descriptions together with its figure generation tools, which consist of a library of predefined plotting functions (e.g., `plot_inclined_plane`). All figure generation functions follow a standardized format: each begins with `def generate_figure(...)` and takes symbolic input variables as arguments, producing a rendered image saved as `figure.png`.

Execution. Each generated figure function is executed by substituting concrete values from the *Input Map*, verifying that the code runs without error. The function is expected to save the rendered diagram as `figure.png`. All final examples were reviewed to ensure consistency between the question, code, and figure.

Substitution and Instance Generation. A key strength of our pipeline is its modular and dynamic structure. For any parameterized or paraphrased question, we systematically substitute values across all components, problem text, reasoning steps, solution code, and figure-generation code, to produce a fully consistent and executable problem instance. Built on a symbolic foundation, the pipeline supports the scalable generation of diverse, scientifically grounded examples. Figure 1 shows an end-to-end instance produced through this process. Additional examples are provided in Appendix F.

Dataset Distribution. To characterize the resulting dataset, PRiSM comprises 24,750 multimodal problem instances spanning Mathematics and Physics. Table 2 details the distribution across domains and subtopics. For further details on

Table 2 Distribution of problem instances in PRiSM.

| Mathematics (18.18%) | Physics (81.82%) |
|--------------------------------|-----------------------|
| Calculus: 7.27% | Kinematics: 36.36% |
| Set Theory: 1.82% | Work & Energy: 10.91% |
| Application of Calculus: 9.09% | Thermodynamics: 9.09% |

coverage and reasoning complexity, please see Appendix E.

4 Benchmark Tasks

Our benchmark facilitates systematic and controlled variation of problem instances to comprehensively evaluate multimodal scientific reasoning capabilities. We define five tasks , each designed to assess a specific aspect of multimodal scientific reasoning: numerical generalization, robustness to visual perturbations, reasoning correction, programmatic generalization, and reasoning under ambiguity. This multifaceted evaluation goes beyond final-answer accuracy Yue et al. (2024); Wang et al. (2023); Lu et al. (2022); Ding et al. (2021); Wang et al. (2024), offering deeper insight into model behavior and interpretability. Examples of these tasks appear in Figure 2 and for more examples refer to the appendix F. For each problem instance, we generate $N = 5$ perturbed versions aligned with the task-specific variation to enable a robust and comparative evaluation.

Task I: Robustness to Input Variations. This task evaluates numerical generalization by varying input values and paraphrasing the problem text, while keeping the visual presentation (e.g., handwriting, style) consistent, as illustrated in Figure 2.

Task II: Robustness to Visual Perturbations. Complementary to Task I, this task probes a model’s resilience to variations in the visual input, such as noise and handwriting style, while keeping the input variables constant.

Task III: Reasoning with Correction. This task evaluates a model’s ability to generalize by identifying and correcting errors in step-by-step reasoning. We present models with partially incorrect solutions, which are synthetically generated using LLaMA 4 Maverick AI (2024) to simulate common student reasoning mistakes, as characterized in cognitive science literature Domondon (2025). The errors are categorized into four types: (i) conceptual errors (misapplication of principles), (ii) careless errors (minor arithmetic mistakes), (iii) encoding errors (misinterpretation of the problem statement), and (iv) knowledge gaps errors (responses based on guessing due to knowledge gaps). Examples of such mistakes are illustrated in Figure 2.

Table 3 Performance of different models on Task I-IV across various metrics.

| Task | Metric | Qwen-72B Yang et al. (2024) | LLaMA-4 Maverick Meta AI (2025) | Mistral Medium-3 Mistral AI (2025) | Claude-3.7 Sonnet Anthropic (2024) | Gemini-2.5 Pro DeepMind (2025) | GPT-4o OpenAI (2024) | o4-mini-high OpenAI (2025) |
|------|--------------------|--------------------------------|------------------------------------|---------------------------------------|---------------------------------------|-----------------------------------|-------------------------|-------------------------------|
| I | Overall Accuracy ↑ | 51.4 | 69.2 | 57.6 | 61.6 | 78.2 | 55.1 | 80.6 |
| | TRUE score ↑ | 8.3 | 36.7 | 27.6 | 30.0 | 56.7 | 23.3 | 60.00 |
| | Volatility ↓ | 33.3 | 18.3 | 15.5 | 28.3 | 6.7 | 26.7 | 10.0 |
| | TFR ↓ | 13.3 | 10.0 | 18.9 | 11.8 | 11.7 | 10.1 | 8.3 |
| II | Overall Accuracy ↑ | 56.3 | 69.5 | 49.8 | 57.7 | 80.3 | 55.53 | 79.8 |
| | TRUE score ↑ | 23.3 | 48.3 | 24.1 | 28.3 | 63.3 | 30.0 | 61.7 |
| | Volatility ↓ | 30.0 | 16.7 | 22.4 | 31.7 | 5.0 | 26.7 | 8.3 |
| | TFR ↓ | 18.3 | 10.1 | 24.1 | 15.0 | 10.0 | 18.3 | 13.3 |
| III | Overall Accuracy ↑ | 36.3 | 65.1 | 42.7 | 59.9 | 79.7 | 59.30 | 79.7 |
| | TRUE score ↑ | 6.7 | 31.8 | 6.7 | 28.3 | 56.7 | 23.3 | 50.0 |
| | Volatility ↓ | 36.7 | 20.0 | 48.3 | 30.0 | 3.3 | 28.3 | 5.0 |
| | TFR ↓ | 15.0 | 11.7 | 15.0 | 10.0 | 10.2 | 15.0 | 8.3 |
| IV | Overall Accuracy ↑ | 13.7 | 15.4 | 5.3 | 17.3 | 15.2 | 17.9 | 21.4 |
| | TRUE score ↑ | 8.9 | 7.1 | 8.9 | 12.5 | 10.7 | 10.7 | 19.6 |
| | Volatility ↓ | 3.6 | 7.1 | 5.4 | 1.8 | 1.9 | 5.4 | 3.6 |
| | TFR ↓ | 75.1 | 78.5 | 75.0 | 78.6 | 78.6 | 76.8 | 76.8 |

Task IV: Programmatic Solution Synthesis. This task assesses a model’s ability to generate executable Python code for solving multimodal scientific problems. Each question is presented in a generic form with input variables expressed symbolically. This formulation evaluates the model’s ability to perform symbolic reasoning and generalize solution strategies using executable Python code, assessing its proficiency in tool use via Python code Schick et al. (2023); Gao et al. (2023); Chen et al. (2021). We execute the generated Python code on different variations of input values and compare the outputs with ground-truth answers obtained from reference Python implementations.

Task V: Reasoning Under Ambiguity. This task evaluates reasoning under uncertainty, a key challenge in real-world scientific problem solving Weiss (2003); Kahneman et al. (1982). We introduce uncertainty by systematically masking 20-30% of input variables and replacing them with symbolic placeholders. We analyze whether models (i) defer or seek clarification, (ii) maintain symbolic representations, or (iii) make assumptions without sufficient information. These behaviors offer valuable insights into model reliability when dealing with incomplete information.

5 Evaluation

5.1 Evaluation Metrics

To evaluate the consistency and robustness of different models across problem variations, we report multiple metrics beyond standard accuracy, drawing inspiration from recent efforts such as SCORE Nalbandyan et al. (2025).

Overall Accuracy: The overall percentage of correctly answered problem instances across all variations. Overall Accuracy:

$$\frac{\sum_{p \in \mathcal{P}} \sum_{v \in \mathcal{V}_p} \mathbb{I}[\hat{y}(p, v) = y(p, v)]}{\sum_{p \in \mathcal{P}} |\mathcal{V}_p|}$$

where \mathcal{P} is the set of all problems, \mathcal{V}_p is the set of variations for problem p , $\hat{y}(p, v)$ is the model prediction, $y(p, v)$ is the ground truth, and $\mathbb{I}[\cdot]$ is the indicator function.

TRUE Score (Total Reliable Understanding Evaluation): To assess the robustness of models in consistently solving each problem across its variations, we define the *TRUE Score* as the proportion of problems for which the model achieves at least $t\%$ accuracy across its associated variations. In our experiments, we set $t = 90$.

$$\text{TRUE}_{t\%} = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \mathbb{I} \left[\frac{1}{|\mathcal{V}_p|} \sum_{v \in \mathcal{V}_p} \mathbb{I}[\hat{y}(p, v) = y(p, v)] \geq t \right] \times 100\%$$

Volatility Rate: The fraction of problems where the model’s accuracy across variations falls between 40% and 60%. This metric reflects the model’s sensitivity to perturbations, indicating inconsistent behavior across semantically equivalent inputs. Volatility Rate:

$$\frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \mathbb{I} \left[0.4 \leq \frac{\sum_{v \in \mathcal{V}_p} \mathbb{I}[\hat{y}(p, v) = y(p, v)]}{|\mathcal{V}_p|} \leq 0.6 \right] \times 100\%$$

Total Failure Rate (TFR): The percentage of problems where the model fails to solve *any* variation. A high TFR indicates fundamental reasoning gaps, where the model is unable to solve the problem, regardless of its phrasing or visualization.

$$\text{TFR} = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \mathbb{I} [\forall v \in \mathcal{V}_p, \hat{y}(p, v) \neq y(p, v)] \times 100\%$$

5.2 Experimental Results

Table 3 presents a quantitative evaluation of model performance across Tasks I–IV, focusing on robustness, reasoning fidelity, and code generation efficacy. Task V, addressing model behavior under ambiguous or underspecified inputs, is assessed qualitatively via an LLM-as-judge framework. Below, we highlight key insights from each task, emphasizing strengths and limitations across models.

Tasks I–III: Our analysis of Tasks I (Robustness to Input Variations), II (Robustness to Visual Perturbations), and III (Reasoning with Correction) reveals consistent trends in how different models handle variations in input and reasoning.

For **Task I**, while overall accuracy suggests strong performance for models like Gemini-2.5 Pro and o4-mini-high (78–80%), the TRUE score provides a more detailed measure of robustness. o4-mini-high demonstrates strong resilience with a TRUE score of 60%, indicating consistent correctness across input variations for a majority of evaluated problem instances. In contrast, Qwen-72B, despite achieving an overall accuracy exceeding 50%, exhibits a significantly lower TRUE score of approximately 8% for Task I. This disparity underscores its vulnerability to even subtle rephrasing or variations in input variables.

Consistent with Task I, Gemini-2.5 Pro and o4-mini-high exhibit strong TRUE scores (63.33% and 61.67%, respectively) in **Task II**, highlighting their robustness to visual perturbations. In contrast, Mistral Medium-3 demonstrates a notable disparity between overall accuracy and TRUE score, suggesting a sensitivity to visual variations. For further insights, we conducted a detailed analysis to identify the errors and incorrect answers produced by models. These findings are discussed in detail in Appendix A.

Task III reveals a significant performance decline in Qwen-72B and Mistral Medium-3, resulting in substantially lower TRUE scores for these models. This performance disparity highlights a specific vulnerability of Qwen-72B and Mistral Medium-3 to the intricacies of the reasoning with correction task, contrasting with the more consistent performance observed in other evaluated models. In addition, we randomly selected 100 samples for detailed analysis of this task. The results can be found in Appendix B.

For **Task IV**, Programmatic Solution Synthesis, we assessed models’ ability to generate executable Python code using SI-unit inputs, as none of the models consistently handled unit conversion. High failure rates were observed across models due to (i) syntactic errors (e.g., missing imports, improperly defined parameters), and (ii) conceptual mistakes (e.g., incorrect symbolic derivations or variable substitutions). Among all models, o4-mini-high exhibited the most consistent and accurate performance in generating executable code. For future work, we intend to investigate these metrics after enabling the model to self-correct by providing them with the execution tool to run their code.

Finally, for **Task V** (Ambiguity Handling), we evaluated models under incomplete or ambiguous inputs, instructing them to request clarification when necessary. Using LLaMA-4 Maverick as an LLM-based judge, we assessed whether models (i) deferred answers or asked clarifying questions, (ii) used symbolic placeholders, or (iii) made unjustified assumptions. Despite explicit instructions, models frequently ignored ambiguity, proceeding with unsupported completions. Symbolic reasoning occurred in only 4–5% of cases, and deferral in just 3–4%. This highlights a fundamental limitation of current VLMs, where they often prioritize providing an answer even when such an answer is not logically supportable without further clarification. Since our judge may introduce some bias, we performed a manual review of 100 randomly selected examples from several models to verify and deepen our understanding of ambiguity handling. This qualitative analysis confirmed that models seldom ask clarifying questions and often make unsupported assumptions based on visual cues or default values, frequently using phrases such as “for simplicity, assume...”. While this enables continued problem-solving, it can mislead users into thinking these assumptions were part of the original problem. Additionally, models often produce partially simplified symbolic expressions that obscure variable dependencies. These observations

reinforce the conclusion that current multimodal models tend to prioritize generating answers over managing uncertainty, underscoring the importance of developing better clarification mechanisms. Full details are provided in Appendix C.

6 Conclusion and Future Work

We proposed PRiSM, a novel benchmark designed to rigorously evaluate scientific reasoning in vision-language models. By employing a scalable, agent-based pipeline (PrismAgent), we have constructed a dynamic and multimodal dataset comprising 24,750 university-level problems. PRiSM is characterized by its emphasis on structured outputs, including executable Python code for ground truth generation and verification, and detailed, step-by-step reasoning. Our comprehensive evaluation, encompassing five targeted tasks, reveals significant limitations in current VLMs' ability to handle the complexities of scientific reasoning.

While current iterations of PRiSM focus on problems within physics and mathematics, future work will broaden the scope to encompass other scientific fields and incorporate additional multimodal sources like audio and video. These extensions will augment visual reasoning challenges and enhance coverage of real-world scenarios. By expanding in these directions, PRiSM aims to advance the rigorous assessment of VLMs and foster progress toward robust, generalizable scientific reasoning.

References

- Meta AI. Llama 4: Advancing open multimodal intelligence. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>, 2024. Accessed: 2025-05-10.
- Anthropic. Introducing the Claude 3 Model Family. <https://www.anthropic.com/news/clause-3-family>, March 2024. Launch post for Claude 3 Sonnet; accessed 2025-05-12.
- Daman Arora, Himanshu Gaurav Singh, et al. Have llms advanced enough? a challenging problem solving benchmark for large language models. *arXiv preprint arXiv:2305.15074*, 2023.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Michelene TH Chi, Paul J Feltovich, and Robert Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive science*, 5(2):121–152, 1981.
- Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.
- Google DeepMind. Gemini 2.5: Our Most Intelligent AI Model. <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/>, March 2025. Gemini 2.5 Pro announcement; accessed 2025-05-12.
- Frances Ding, Moritz Hardt, John Miller, and Ludwig Schmidt. Retiring adult: New datasets for fair machine learning. *Advances in neural information processing systems*, 34:6478–6490, 2021.
- Christian Domondon. Analyzing the errors of stem students in solving basic calculus problems. *Diversitas Journal*, 10(1), 2025.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.
- Daniel Kahneman, Paul Slovic, and Amos Tversky. *Judgment under uncertainty: Heuristics and biases*. Cambridge university press, 1982.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.
- Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40:e253, 2017.
- Pan Lu, Swaroop Mishra, Tanglin Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. Learn to explain: Multimodal reasoning via thought chains for science question answering. *Advances in Neural Information Processing Systems*, 35:2507–2521, 2022.
- Meta AI. Introducing LLaMA 4: Advancing Multimodal Intelligence. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>, April 2025. Accessed: 2025-05-12.
- Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. <https://doi.org/10.7717/peerj-cs.103>.
- Mistral AI. Mistral-medium-3: Medium is the new large. <https://mistral.ai/news/mistral-medium-3/>, April 2025. Accessed: 2025-05-12.
- Grigor Nalbandyan, Rima Shahbazyan, and Evelina Bakhturina. Score: Systematic consistency and robustness evaluation for large language models. *arXiv preprint arXiv:2503.00137*, 2025.
- Allen Newell and Herbert Simon. Human problem solving. *Prentice-Hall*, 1, 1972.
- OpenAI. Chatgpt plugins, 2023. <https://openai.com/index/chatgpt-plugins/>. Accessed: 2025-03-18.

OpenAI. Hello gpt-4o. <https://openai.com/index/hello-gpt-4o/>, May 2024. Model launch page; accessed 2025-05-12.

OpenAI. Model release notes – openai o3. <https://help.openai.com/en/articles/9624314-model-release-notes>, April 2025. Release notes section "OpenAI o3 and o4-mini"; accessed 2025-05-12.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Pint Developers. Pint: Python units library, 2025. <https://pint.readthedocs.io/>.

George Polya. How to solve it: A new aspect of mathematical method. In *How to solve it*. Princeton university press, 2014.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36: 53728–53741, 2023.

Matthew Renze and Erhan Guven. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Liangtai Sun, Yang Han, Zihan Zhao, Da Ma, Zhennan Shen, Baocai Chen, Lu Chen, and Kai Yu. Scieval: A multi-level large language model evaluation benchmark for scientific research. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19053–19061, 2024.

Angelina Wang, Aaron Hertzmann, and Olga Russakovsky. Benchmark suites instead of leaderboards for evaluating ai fairness. *Patterns*, 5(11), 2024.

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. Scibench: Evaluating college-level scientific problem-solving abilities of large language models. *arXiv preprint arXiv:2307.10635*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Charles Weiss. Expressing scientific uncertainty. *Law, Probability and Risk*, 2(1):25–46, 03 2003. ISSN 1470-8396. doi: 10.1093/lpr/2.1.25. <https://doi.org/10.1093/lpr/2.1.25>.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL <https://arxiv.org/abs/2305.10601>, 3, 2023.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, et al. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9556–9567, 2024.

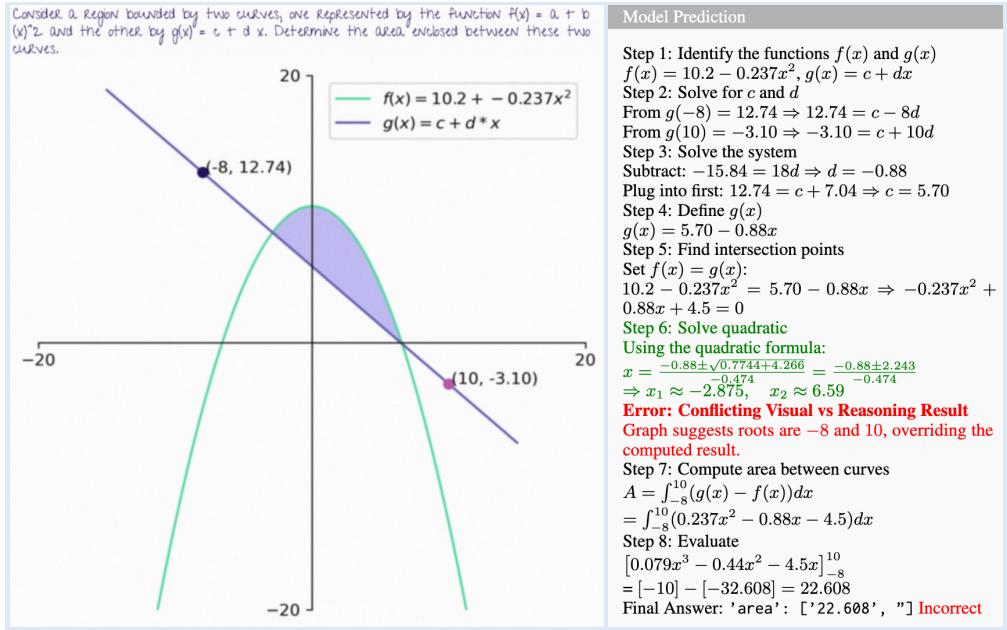


Figure 4 Illustration of modality conflict where the model’s correct algebraic solution is overridden by a misleading visual cue.

Appendix

A Analysis: Robustness to Text and Visual Perturbations (Task I & II)

To gain deeper insight into model failure modes within PRISM, we randomly sampled 500 examples and conducted a thorough qualitative analysis of the model outputs. Although our initial experiments focused on the output of Maverick model, we observed similar behaviors across other models such as Gemini-2.5 Pro and Qwen-72B, indicating these issues are broadly representative.

Below, we summarize the key failure categories:

Modality Conflict Despite correctly solving problems through algebraic or symbolic reasoning, the model sometimes overrides these solutions due to ambiguity or noise present in the visual modality. This reflects a lack of a robust verification mechanism to reconcile the perceptual (image-based) input with the symbolic reasoning process. As a result, the model often places undue trust in the visual information, even when it contradicts more precise calculations.

Figure 4 illustrates an example of this issue, with the model’s prediction displayed in the gray box. Although the model correctly computes the intersection points of two functions algebraically, it ultimately produces incorrect results by relying on a misleading visual interpretation of the graph.

Ambiguity-Induced Errors These errors arise from ambiguous or unclear visual features, such as misreading handwritten digits or overlooking critical decimal points. This issue stems from insufficient robustness in interpreting noisy or poorly rendered visual inputs, leading to incorrect numeric or symbolic interpretations that are not reconciled with the broader problem context.

As shown in Figure 5, the handwritten value “ $L = 6.31$ m” was initially misread as “ 0.31 m” due to a curled digit and a faint decimal point, leading the model to an incorrect interpretation. However, when presented with a zoomed-in version of the image, the model recognized and corrected the error.

Visual Misreading Errors These errors occur when the model misreads numerical values, especially in dense annotations or when dealing with large numbers.

A linear source of sound, such as an electric spark, emits a pulse of sound along a straight line of length $L = 6.31$ m. The power of this acoustic emission is $P_s = 11100.0$ W. (a) What is the intensity I of the sound at a distance $R = 8.95$ m from the spark? (b) At what rate P_d does the sound energy intercept an acoustic detector with an area $A_d = 0.000233 \text{ m}^2$, aimed at the spark and located at a distance $R = 8.95$ m from it? Here are constants:
 $\{\}$

Figure 5 Example of ambiguity in handwritten digits leading to misinterpretation.

For example, in the question shown in Figure 6, the model interpreted the oscillator frequency as $f = 1,000,000.0$ Hz, whereas the correct value was $f = 10,000,000.0$ Hz. The additional zero was visually subtle, and the model failed to verify the magnitude against contextual cues.

A particle accelerator, specifically a cyclotron, is set to operate at an oscillator frequency of 10000000.0 Hz and has a dee radius of 0.603 m. (a) Determine the magnitude of the magnetic field required to accelerate deuterons in this cyclotron. The mass of a deuteron is 3.34e-27 kg (twice the mass of a proton). Here are constants: {'m': 3.34e-27 kg, 'q': 1.6e-19 C}

Figure 6 Example of ambiguous digit rendering leading to incorrect frequency interpretation.

Multi-Step Reasoning Gaps Even when the model performs step-by-step reasoning, it often fails in problems requiring decomposition of a single step into multiple intermediate hops. Attempting to collapse these multi-hop logical steps into one frequently causes the model to skip key transformations, particularly during unit conversions, chained exponential operations, or algebraic simplifications, resulting in incorrect final answers.

For example, in the following calculation, the model initially provides clear, step-by-step reasoning. However, instead of maintaining this detailed approach, it abruptly collapses the multi-hop process, such as computing exponentials and multiplications, into a single step and outputs a numerical answer without explanation.

Model Prediction

Step 3: Perform the calculation for K

$$\begin{aligned} K &= \frac{1}{4\pi \times 8.85 \times 10^{-12}} \cdot \frac{(1.6 \times 10^{-19})^2}{2 \times 10^{-15}} \\ &= \frac{1}{4\pi \times 8.85 \times 10^{-12}} \cdot \frac{2.56 \times 10^{-38}}{2 \times 10^{-15}} \\ &= \frac{1}{1.113 \times 10^{-10}} \cdot 1.28 \times 10^{-23} \end{aligned}$$

Incorrect:

$$K = 8.99 \times 10^{-14} \cdot 1.28 \times 10^{-23} = 1.15 \times 10^{-36} \text{ J}$$

Expected:

$$K = 8.99 \times 10^9 \cdot 1.28 \times 10^{-23} = 1.15 \times 10^{-13} \text{ J}$$

Error Explanation: The model collapses multiple steps, approximating $1/(1.113 \times 10^{-10})$, converting it to the Coulomb constant 8.99×10^9 , and then performing the multiplication, all in a single step. In doing so, it incorrectly substitutes 8.99×10^{-14} instead of 8.99×10^9 , leading to a drastic exponent error and an incorrect final result.

Numerical Precision Models often perform numerically correct computations but fail to adjust precision appropriately for the problem context. Unlike humans, who infer the number of significant digits or reason about precision based on the physical setup, models frequently over approximate or under approximate without justification. This leads to issues such as numerical instability when subtracting large, nearly equal values (e.g., computing $\Delta r = r_1 - r_2$ where $r_1 \approx r_2$), resulting in inaccurate final results despite correct symbolic expressions. Models also make errors in exponentiation calculations, for instance, computing 1.79^5 and incorrectly reporting 24.76 where the correct answer is 18.37.

Conceptual or Formula Misapplication This category includes errors such as using incorrect formulas. For example, in physics problem applying $M = \mu_0 N_1 N_2 R^2 / l$ instead of the correct $M = \mu_0 N_1 N_2 A / l$ or misapplying the Hall voltage formula by using the wrong variable for thickness or length. Models also confuse physical setups, such as assuming standing wave modes for a string fixed at both ends when it is actually fixed at only one end. Additionally, there are mistakes in interpreting circuits, like mixing up series and parallel configurations despite visual cues.

B Analysis: Reasoning with Correction (Task III)

To better understand model failure modes on the *Reasoning with Correction* task, we conducted a qualitative analysis of 100 randomly sampled examples output from Qwen-72B and Gemini-2.5 Pro. We categorized the models' responses into three main types:

1. **Silent Correction:** The model ignores the student's error and provides a corrected solution without referencing the mistake.

Examples:

- "Let's continue solving the problem step by step..."
- "Ignore the student's work and start from scratch."

2. **Uncritical Acceptance:** The model accepts the student's reasoning without verifying correctness.

Examples:

- "The student's approach is correct. We need to complete the calculations."
- "The student calculated electric force correctly but hasn't completed the work and energy part."

3. **Explicit Error Identification and Correction:** The model identifies the student's error, explains it, and then provides the corrected solution.

Examples:

- “The student made a mistake in applying Ohm’s law...”
- “There is an algebraic error in the student’s solution. Here is the corrected version...”

Table 4 summarizes the results of our quantitative analysis. Gemini-2.5 Pro demonstrates a notably higher propensity to explicitly detect and correct errors compared to Qwen-72B. In addition, other behaviors such as silent correction and uncritical acceptance indicate missed opportunities for offering valuable, instructive feedback. Therefore, this task evaluates not only the accuracy of model responses but also their diagnostic reasoning and ability to provide meaningful guidance.

| Behavior Type | Qwen-72B | Gemini-2.5 Pro |
|-----------------------|----------|----------------|
| Explicit Correction | 62% | 84% |
| Silent Correction | 24% | 6% |
| Uncritical Acceptance | 14% | 10% |

Table 4 Distribution of observed behaviors in the Reasoning with Correction task. Note that high explicit correction rates do not guarantee correctness of the final answer.

C Analysis: Ambiguity Handling (Task V)

To better understand the task and common failure modes of ambiguity task, we manually reviewed 100 random samples from OpenAI’s o4-mini-high, Gemini-2.5 Pro (closed-source models), and Qwen-72B (open-source model). Our qualitative analysis highlights why models rarely ask clarifying questions, even when explicitly prompted:

1. **Unjustified Assumptions from Visual Context.** Models often infer missing parameters directly from the image instead of requesting clarification. For example, in a question missing horizontal coordinates x_1 and x_2 , Qwen-72B incorrectly assumed $x_1 = x_2$, relying only on vertical separation cues in the figure, resulting in an invalid assumption.
2. **Assumptions Without Explicit Clarification.** When unable to visually infer values, models proceed with default assumptions, using phrases such as “for simplicity, assume...”. While this enables continued problem-solving, it can mislead users into thinking these assumptions were part of the original problem.
3. **Partial or Incomplete Symbolic Simplifications.** Models like Gemini-2.5 Pro and o4-mini-high sometimes stop at symbolic answers involving unsimplified variables, which, although not incorrect, obscure the final dependency structure. For instance, if the target is y and the model outputs $y = \frac{A}{c}$, but $A = a \times c$, the fully simplified answer should be $y = a$. Stopping early yields a less informative and potentially misleading result.

Models tend to infer or assume missing information rather than ask clarifying questions, reflecting a core limitation where multimodal models prioritize solution completion over managing uncertainty. Our findings underscore the need for explicit deferral or clarification capabilities in VLMs.

D Predefined Plotting Functions

Our benchmark utilizes a set of modular Python plotting utilities (mainly based on `matplotlib`) to standardize figure generation in math and physics problems. These functions, collected in `figure_utils.py`, abstract low-level drawing, enabling the agent to produce consistent and accurate figures efficiently.

Representative functions include:

- `add_inclined_surface_with_angle(ax, ...)`
- `add_block_on_incline(ax, ...)`
- `draw_horizontal_resistor(ax, ...)`

Development was iterative, driven by dataset analysis and aided by LLMs to identify common diagram components. All generated figure code was manually reviewed for correctness.

E Appendix: Benchmark Statistics

Domain Coverage The physics domain covers 450 concepts, with core topics such as energy (2.67%), force (2.00%), velocity (2.00%), and kinetic energy (1.78%). Additional frequent topics include acceleration, mass, electric field, momentum, work, magnetic field, potential energy, torque, Coulomb force, power, and wave motion. The long tail features concepts like gravitational effects, moment of inertia, Faraday’s law, projectile motion, and lens equation (each approximately 0.22%).

In the mathematics domain, which includes 110 concepts, top concepts are area (5.45%), function (5.45%), and interval (3.64%). Other covered topics include slope, algebraic expressions, equations, symmetry, triangles, circles, inequalities, coordinate geometry, ratios, integration, differentiation, Venn diagrams, and set theory.

Reasoning Complexity: Problems in PRiSM require multi-hop, structured reasoning as reflected in the problem step counts: mean steps per problem is 6.91, with a minimum of 4 and maximum of 11 steps, and a standard deviation of 1.75. This indicates that our benchmark consistently demands deep reasoning rather than shortcut heuristics.

Necessity and Role of Images We performed a detailed analysis of how variables are distributed across the different modalities, including text and figures, within our dataset to understand the unique contributions of each modality and to demonstrate the necessity of multimodal reasoning.

- **Problem Inputs:** Number of input variables required to solve the problem (from all modalities).
- **Problem Outputs:** Number of output variables or answers expected for the problem.
- **Figure Inputs:** Number of variables present in the figure (including those also in text).
- **Text Inputs:** Number of variables present in the text (including those also in figures).
- **Figure-only Inputs:** Number of variables appearing exclusively in the figure, not in text.
- **Text-only Inputs:** Number of variables appearing exclusively in the text, not in figures.

| Variable Type | Min | Max | Mean | Std Dev |
|--------------------|-----|------|------|---------|
| Problem Inputs | 1.0 | 12.0 | 4.43 | 1.65 |
| Problem Outputs | 1.0 | 8.0 | 2.20 | 1.81 |
| Figure Inputs | 1.0 | 8.0 | 3.50 | 1.40 |
| Text Inputs | 0.0 | 11.0 | 3.39 | 1.74 |
| Figure-only Inputs | 0.0 | 8.0 | 0.62 | 1.50 |
| Text-only Inputs | 0.0 | 9.0 | 0.52 | 1.33 |

Table 5 Descriptive statistics of variable counts across modalities in PRiSM.

A substantial portion of problems require information extracted jointly from both figures and text, highlighting the complementary roles of these modalities. Importantly, a significant number of variables appear *exclusively* in the figures, underscoring the indispensable role of visual reasoning. Similarly, certain variables occur only in the textual modality, validating the necessity for effective multimodal fusion to support comprehensive model reasoning. Additionally, due to the dynamic nature of our benchmark, overlapping variables may be included in any modality as needed.

F Appendix: Examples from PRiSM dataset

In this section, we provide more examples from the PRiSM dataset.

F.1 Python Code Scaffolding

An excerpt of the prompt used to guide the agent in generating fully generalized Python functions and paraphrasing of the problem is shown in F.1.

Example 1

Question

Consider a circuit consisting of three capacitors, with capacitances C_1 , C_2 , and C_3 , connected in a combination of series and parallel configurations. When a potential difference of V volts is applied across the entire circuit, determine the net capacitance of the combination and find the charge and voltage across each individual capacitor.

Reasoning Steps

To solve this, first, find the equivalent capacitance of the parallel connection between C_2 and C_3 , denoted as C_{23} , which is simply the sum of C_2 and C_3 :

$$C_{23} = C_2 + C_3.$$

Then, consider the entire circuit as a series connection between C_1 and C_{23} . The net capacitance C of this series connection can be found using the relation:

$$\frac{1}{C} = \frac{1}{C_1} + \frac{1}{C_{23}}.$$

Since capacitors in series carry the same charge, we can use the applied voltage 12.8 V and the net capacitance C to find the charge Q_1 on C_1 :

$$Q_1 = C \cdot 12.8 \text{ V}.$$

The voltage across C_1 , denoted V_1 , is given by:

$$V_1 = \frac{Q_1}{C_1}.$$

The voltage across the parallel combination of C_2 and C_3 , denoted V_{23} , is then:

$$V_{23} = 12.8 \text{ V} - V_1.$$

Because C_2 and C_3 are in parallel, they share the same voltage:

$$V_2 = V_3 = V_{23}.$$

This allows us to calculate the charges:

$$Q_2 = C_2 \cdot V_2, \quad Q_3 = C_3 \cdot V_3.$$

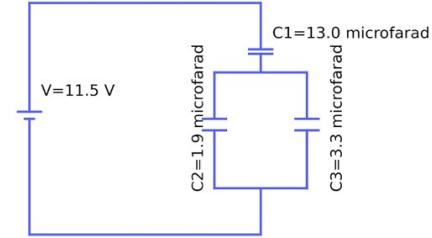
The charge on the parallel combination, Q_{23} , is the sum:

$$Q_{23} = Q_2 + Q_3.$$

This confirms the consistency of our approach. By following these steps, we can determine the net capacitance of the combination, and the charge and voltage across each capacitor.

Input variables used by the Python code to compute the ground truth

```
{
    "generated_variables": {
        "C1": [13.0, "microfarad"], // Capacitance of C1
        "C2": [1.9, "microfarad"], // Capacitance of C2
        "C3": [3.3, "microfarad"], // Capacitance of C3
        "V": [11.5, "V"] // Applied voltage
    },
    "constant_values": {},
    "python_result": {
        "C": 3.71e-06 farad, // Net capacitance in Farads
        "Q1": 4.27e-05 C, // Charge on C1
        "Q2": 1.56e-05 C, // Charge on C2
        "Q3": 2.71e-05 C, // Charge on C3
        "V1": 3.29 V, // Voltage across C1
        "V2": 8.21 V, // Voltage across C2
        "V3": 8.21 V // Voltage across C3
    }
}
```



Python code

```
import sympy as sp
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity

def capacitor_combination(C1, C2, C3, V):
    # Convert inputs to Pint quantities
    C1 = Q_(C1).to(ureg.farad) # Ensure farad
    C2 = Q_(C2).to(ureg.farad) # Ensure farad
    C3 = Q_(C3).to(ureg.farad) # Ensure farad
    V = Q_(V).to(ureg.volt) # Ensure volt

    C1 = C1.magnitude
    C2 = C2.magnitude
    C3 = C3.magnitude
    V = V.magnitude

    # Calculate net capacitance of parallel connection C2 and C3
    C23 = C2 + C3

    # Calculate net capacitance of series connection C1 and C23
    C = (1 / C1 + 1 / C23)**-1

    # Calculate charge Q1
    Q1 = C * V

    # Calculate voltage V1
    V1 = Q1 / C1

    # Calculate voltage V23
    V23 = V - V1

    # Calculate charges Q2 and Q3
    Q2 = C2 * V23
    Q3 = C3 * V23

    return {
        'C': C,
        'Q1': Q1,
        'Q2': Q2,
        'Q3': Q3,
        'V1': V1,
        'V2': V23,
        'V3': V23
    }
```

Figure 7 Example 1 from PRiSM showing input variation and structured reasoning.

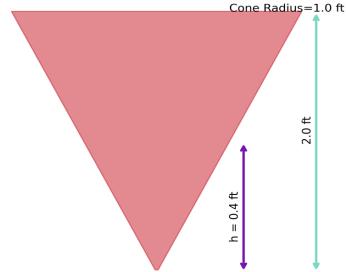
Example 2

Textual Variations of the Question

Text Variation I) A liquid is flowing out of a cone-shaped container at a rate of -0.026 ft³/sec. The container has a height of H and a radius of r at its widest point. What is the rate at which the liquid's surface is descending when its height is h?

Text Variation II) Consider a cone-shaped container with a height of H and a radius of 1.0 ft at its top, being drained at a rate of -0.026 ft³/sec. How is the water level in the container changing when the water reaches a height of h?

Text Variation III) A cone-shaped container, with a height of H and a top radius of r, is being drained at a rate of -0.026 ft³/sec. We are tasked with finding the rate at which the water level is changing when the water height is 0.4 ft.



Python code

```
import sympy as sp
from pint import UnitRegistry

ureg = UnitRegistry()
Q_ = ureg.Quantity

def calculate_height_change(radius, height, h, dV_dt):
    # Convert inputs to Pint quantities
    h = Q_(h).to(ureg.foot) # Ensure ft
    height = Q_(height).to(ureg.foot) # Ensure ft
    radius = Q_(radius).to(ureg.foot) # Ensure ft
    dV_dt = Q_(dV_dt).to(ureg.foot**3 / ureg.second) # # E

    h = h.magnitude
    dV_dt = dV_dt.magnitude
    radius = radius.magnitude
    height = height.magnitude

    # Define symbolic variables
    pi = sp.pi

    # Calculate dh/dt
    if h == 0:
        dh_dt = 0
    else:
        dh_dt = dV_dt / (((radius / height)**2) * pi * h**2)

    return {
        'dh_dt': dh_dt.evalf()
    }
```

Input variables used by the Python code to compute the ground truth

Input variables to the python code to find the ground truth:

```
{
  "generated_variables": {
    "h": [0.4, "ft"], // Current water height
    "H": [2.0, "ft"], // Total cone height
    "R": [1.0, "ft"], // Top radius of cone
    "dV_dt": [-0.026, "ft^3/sec"] // Rate of volume change
  },
  "constant_values": {},
  "python_result": {
    "dh_dt": -0.207 // Rate of water height change
  }
}
```

Reasoning Steps

1. Identify Relevant Concepts

- We are dealing with a *cone-shaped container* with changing water levels, so we'll apply the concept of **related rates** and the formula for the volume of a cone.
- The goal is to find the rate of change of the water height:

$$\frac{dh}{dt},$$

when the water height is

$$h = 0.4 \text{ ft}.$$

- Known values:

$$\frac{dV}{dt} = -0.026 \text{ ft}^3/\text{sec}, \quad H = 2.0 \text{ ft}, \quad R = 1.0 \text{ ft}, \quad h = 0.4 \text{ ft}.$$

2. Set Up the Problem

- The volume of water in the cone is:

$$V = \frac{1}{3}\pi r^2 h,$$

where r is the radius of the water surface and h is its height.

- From similar triangles:

$$\frac{r}{h} = \frac{R}{H} \Rightarrow r = \frac{hR}{H}.$$

- Substituting into the volume formula:

$$V = \frac{1}{3}\pi \left(\frac{hR}{H}\right)^2 h = \frac{\pi R^2}{3H^2} h^3.$$

3. Execute the Solution

- Differentiate both sides with respect to time t:

$$\frac{dV}{dt} = \frac{\pi R^2}{H^2} h^2 \frac{dh}{dt}.$$

- Solving for $\frac{dh}{dt}$:

$$\frac{dh}{dt} = \frac{(dV/dt) \cdot H^2}{\pi R^2 h^2}.$$

4. Evaluate Your Answer

- The solution is reasonable: as water drains, the water level drops faster or slower depending on the remaining height and cone shape.
- The derived expression for $\frac{dh}{dt}$ captures the influence of all relevant parameters.

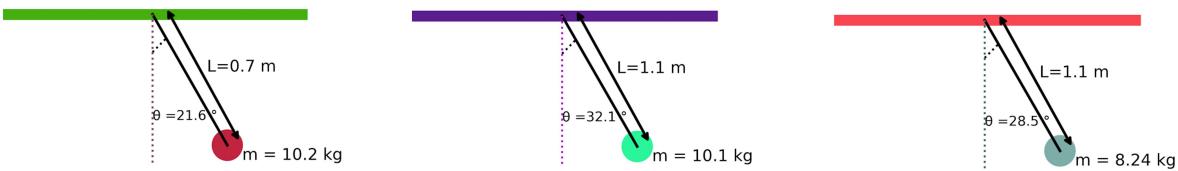
Figure 8 Example 2 from PRiSM showing reasoning correction.

Example 3

Question

A pendulum consisting of a particle of m attached to a massless string of length L is initially displaced by an angle θ from the vertical. Assuming the only force acting on the particle is gravity and neglecting air resistance, what is the speed of the particle when it reaches the bottom of its swing? Here are the constants: $g=9.8\text{m/s}^2$.

Different input variations of the problem are illustrated entirely in the figure, with no numerical values provided in the question text.



Input variables used by the Python code to compute the ground truth

```
{
  "generated_variables": {
    "L": [0.7, "m"],
    "m": [10.2, "kg"],
    "theta": [21.6, "°"]
  },
  "constant_values": {
    "g": ["9.8", "m/s²"]
  },
  "python_result": {
    "v": 0.982 m/s
  }
}

{
  "generated_variables": {
    "L": [1.1, "m"],
    "m": [10.1, "kg"],
    "theta": [32.1, "°"]
  },
  "constant_values": {
    "g": ["9.8", "m/s²"]
  },
  "python_result": {
    "v": 1.82 m/s
  }
}

{
  "generated_variables": {
    "L": [1.1, "m"],
    "m": [8.24, "kg"],
    "theta": [28.5, "°"]
  },
  "constant_values": {
    "g": ["9.8", "m/s²"]
  },
  "python_result": {
    "v": 1.62 m/s
  }
}
```

Python Code

```
import sympy as sp
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity

def speed_at_lowest_point(L, theta, g):
    # Convert inputs to Pint quantities
    L = Q_(L).to(ureg.meter) # Ensure meters
    theta = Q_(theta).to(ureg.degree) # Ensure degrees
    g = Q_(g).to(ureg.meter / ureg.second**2) # Ensure m/s^2

    L = L.magnitude
    theta = theta.to(ureg.radian).magnitude # Convert degrees to radians
    g = g.magnitude

    # Define symbolic variables
    h = L * (1 - sp.cos(theta)) # Vertical height

    # Calculate the speed at the lowest point using conservation of energy
    v = sp.sqrt(2 * g * h)

    return {
        'v': v
    }
```

Reasoning Step

- Identify Relevant Concepts:
 - The problem involves a conservative force (gravity) acting on the particle, with non-conservative forces (like air resistance) neglected.
 - The goal is to find the speed of the particle at the lowest point of its swing.
 - Known quantities include the string length L , the initial angle θ , and the acceleration due to gravity g .
- Set Up the Problem:
 - Use the principle of conservation of mechanical energy: total mechanical energy (kinetic + potential) remains constant.
 - Choose the lowest point of the swing as the reference point where gravitational potential energy is zero.
 - At the release point, the particle starts from rest, so initial kinetic energy is zero.
- Execute the Solution:
 - Apply energy conservation: $K_i + U_i = K_f + U_f$.
 - At the top (initial): $K_i = 0, U_i = mgh$; at the bottom (final): $K_f = \frac{1}{2}mv^2, U_f = 0$.
 - The energy equation simplifies to: $mgh = \frac{1}{2}mv^2$.
 - Solve for v : $v = \sqrt{2gh}$
 - Express h using trigonometry: $h = L(1 - \cos \theta)$
 - Final speed formula: $v = \sqrt{2gL(1 - \cos \theta)}$
- Evaluate Your Answer:
 - The equation shows that as θ increases, h increases, leading to a larger v , which aligns with physical intuition.
 - For small or large angles, the expression behaves consistently with expectations, confirming its validity.

Figure 9 Example 3 from PRISM demonstrating multimodal synthesis.

Prompt Template for Python Code Synthesis

Given a problem, perform the following tasks:

Generate a Python Function:

- The function should take the provided Inputs as arguments.
- It must return only the specified Outputs variables.
- Using the given Inputs and the solution, write a Python function that solves the problem and returns the required Outputs.

Function Requirements:

- Name the function appropriately based on the context of the problem.
- The function should take only the identified Inputs variables as arguments.
- Use **Sympy** for symbolic calculations (e.g., algebraic manipulations, solving equations, simplifying expressions).
- Use **Pint** for all unit handling and conversions.
 - Apply units to variables using Pint (e.g., $v0 = Q_{v0}.to(ureg.meter / ureg.second)$) to ensure correct units.
 - Extract magnitudes using `.magnitude` before performing numerical calculations.
 - Do not manually convert units** (e.g., avoid converting cm to m manually; always use Pint and then get the magnitude).
- Preserve symbolic calculations where possible.
- Utilize SymPy functions such as `solve`, `dsolve`, `integrate`, `diff`, `simplify`, `expand`, and `factor`.
- IMPORTANT:** Ensure that your Python function is fully generalized—do not hardcode any specific variables. All values should be dynamically derived from the provided input parameters.
- If your Python code includes symbolic expressions such as `sp.sin`, `sp.pi`, or `sp.cos`, make sure to apply `.evalf()` so that when variables are substituted, the output is numerical rather than symbolic.
- Always use fully qualified names such as `sp.cos`, `sp.sin`, or `sp.pi` instead of importing symbols directly.

Python Function Output:

- The function should return a **dictionary** where:
 - Keys** correspond to the specified Outputs variable names.
 - Values** are lists of the corresponding parameters.

Output Format:

- Provide the final output in **JSON format**.

Output Format (JSON):

```
import sympy as sp
import numpy as np
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity
def solution(Inputs):
    """ Steps to solve the problem
    return {'ans_1': [ans1, "unit_1"], "ans_2": [ans2, "unit_2"]}
    # Only return the required Outputs variables
Example Output: for Inputs=[v0, g, ytarget] and Outputs=['time_to_target', 'max_height']
import sympy as sp
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity

def projectile_motion(v0, g, y_target):
    # Convert inputs to Pint quantities
    v0 = Q_(v0).to(ureg.meter / ureg.second) # Ensure m/s
    g = Q_(g).to(ureg.meter / ureg.second**2) # Ensure m/s^2
    y_target = Q_(y_target).to(ureg.meter) # Ensure meters

    v0 = v0.magnitude
    g = g.magnitude
    y_target = y_target.magnitude

    # Define symbolic variables
    t = sp.Symbol('t', real=True, positive=True)

    # (1) Solve for time to reach y_target
    y = v0 * t - (1/2) * g * t**2 # Motion equation
    time_solutions = sp.solve(sp.Eq(y, y_target), t)

    # Filter out negative time values
    time_to_target = [sol.evalf() for sol in time_solutions if sol.is_real and sol > 0]

    # (2) Calculate maximum height
    t_max = v0 / g # Time to reach max height (v = 0)
    y_max = v0 * t_max - (1/2) * g * t_max**2

    return {
        'time_to_target': time_to_target,
        'max_height': y_max
    }
```

Prompt Template for Paraphrasing I

Paraphrase the given generic question and solution while ensuring clarity, creativity, and structured refinement. Follow these guidelines:

Important: The list of symbols ending with `_numerical_value` (called `immutable_symbols`) must remain unchanged in both question and solution. Do not add new symbols ending with `_numerical_value`.

Paraphrasing Guidelines:

- Preserve Parameter Names:** Keep all variable names ending with `_numerical_value` unchanged. Do not add units.
- Enhance Readability & Conciseness:** Restructure sentences for clarity and flow; remove redundancies while keeping all key details.
- Refine References & Equations:** Remove mentions of equation numbers or figure references; present equations naturally.
- Introduce Meaningful Variations:** Modify the scenario creatively (e.g., change "car" to "truck") while preserving core concepts, methods, and solution approach.
- Maintain Structural Integrity:** Rewrite the solution using the provided problem-solving template, mirroring original reasoning but in reworded form.
- Make Questions More Abstract:** Ensure the paraphrased question feels like a new problem with improved clarity.

Output Format: Return a JSON object with keys:

```
{
    "Paraphrase_Question": "Paraphrased question with immutable symbols intact.",
    "Paraphrase_Calculation": "Paraphrased solution following the problem-solving template."
}
```

Example Template:

Problem Solving Steps:

- Identify Relevant Concepts
- Set Up the Problem
- Execute the Solution
- Evaluate Your Answer

Note: For all examples, symbols ending with `_numerical_value` are never changed.

Prompt Template for Paraphrasing II

Paraphrase the given generic question and solution while ensuring clarity, creativity, and structured refinement. Follow these updated guidelines to enhance the quality of the paraphrased output:

Most Important: The list of symbols ending with `_numerical_value` (referred to as *immutable_symbols*) must remain unchanged in both the question and solution. Do **not** modify, rename, or introduce any new variables ending with `_numerical_value`.

Paraphrasing Guidelines:

- **Preserve Parameter Names:** All variables ending with `_numerical_value` must remain unaltered. Do not append, rename, or introduce new ones. **Do not include units.**
- **Enhance Readability & Conciseness:** Improve sentence structure for clarity and smooth flow. Eliminate redundancies while preserving all essential information. The text should be natural, precise, and easy to follow.
- **Refine References & Equations:** Remove all references to figure numbers or equations unless explicitly present in the original. Present equations naturally within the flow of the text without reference annotations.
- **Introduce Meaningful Variations:** Modify the context while preserving the underlying mathematical or physical principles. For example, change "car" to "truck" or "rabbit" to "fox." Be creative but maintain logical and conceptual equivalence. Ensure that the original equations and methodology remain applicable.
- **Maintain Structured Reasoning:** Paraphrased solutions must follow the problem-solving template outlined below. Keep the logical structure of the original reasoning intact, but reword and reframe it for improved clarity.
- **Make the Questions More Abstract:** Rephrase the question to sound more generalized or conceptually framed. Avoid overly specific or concrete phrasing if not required.

Problem Solving Template:

- **Read and Understand:** Read the problem thoroughly and ensure full comprehension.
- **Identify Objects and Quantities:** Determine the key objects, knowns, and unknowns. Translate language into mathematical or physical symbols as needed.
- **Apply Relevant Principles:** Identify the laws or equations that govern the situation and plan a logical approach.
- **Select and Apply Equations:** Choose appropriate equations, solve symbolically, and then substitute numerical values.
- **Calculate and Verify:** Perform the computation and verify the accuracy of the result.
- **Check Units:** Ensure dimensional consistency (if applicable), but do not include explicit units in the paraphrased text.

Output Format: Return your output in the following JSON format:

```
{  
    "Paraphrase_Question": "Paraphrased question with immutable symbols intact.",  
    "Paraphrase_Calculation": "Paraphrased solution following the structured problem-solving template."  
}
```

Example:

```
{  
    "Paraphrase_Question": "Consider a mathematical function  $f(x) = c_{numerical\_value}x + d_{numerical\_value}$  defined over a specific range from  $a_{numerical\_value}$  to  $b_{numerical\_value}$ . What is the mean value of this function?",  
    "Paraphrase_Calculation": "To solve this, start by identifying the linear function and the interval it spans. This function forms a trapezoidal area under the curve between  $x = a$  and  $x = b$ . Begin by evaluating the function at the boundaries and then use the formula for the mean value of a function over an interval."  
}
```

Prompt Template for Paraphrasing III

Paraphrase the given generic question and solution while ensuring clarity, creativity, and structured refinement. Follow these enhanced guidelines to improve the quality and instructional clarity of the paraphrased output. **Most Important:** The list of symbols ending with `_numerical_value` (called *immutable_symbols*) must remain unchanged in both the question and solution. Do not modify existing ones or introduce new symbols with this suffix.

Paraphrasing Guidelines:

- **Preserve Parameter Names:** Keep all variable names ending with `_numerical_value` intact. **Do not change these symbols or include units** in either the question or the solution.
- **Enhance Readability & Conciseness:** Restructure sentences to improve clarity and flow. Eliminate redundant phrases while preserving all necessary information. Ensure the paraphrased text is engaging, readable, and logically organized.
- **Refine References & Equations:** Do not refer to figure numbers or equation numbers unless they are explicitly included in the original text. Write equations as part of the natural narrative of the solution.
- **Introduce Meaningful Variations:** Change the context of the problem while maintaining the core concept, underlying method, and the validity of the original equations. For example, replace "asteroid" with "satellite", or "Earth" with "a planet". Ensure logical and scientific consistency in the new context.
- **Maintain Structured Problem Solving:** Rewrite the solution using the specific problem-solving strategy outlined below. The structure should mirror the logical approach of the original solution, but with new wording and context.
- **Make the Questions More Abstract:** Avoid overly specific phrasing unless necessary. Frame the problem in a slightly more general or conceptual manner to give it a fresh presentation.

Structured Problem Solving Template:

- **List Known and Unknowns:** Record all given parameters and identify what is to be solved.
- **Symbolic Solutions:** Solve the problem using algebra before plugging in numerical values.
- **Units and Dimensions:** Ensure that the dimensional units used are consistent and correct throughout the process. Do not write units explicitly in the output.
- **Check Special Cases:** Evaluate limiting or boundary cases (e.g., zero initial speed) to validate that the solution behaves reasonably. Conduct a basic sanity check on the final expression or value.

Output Format: Return your final output in the following JSON format:

```
{  
    "Paraphrase_Question": "Paraphrased question that introduces contextual variation but keeps all  
                           immutable symbols unchanged.",  
    "Paraphrase_Calculation": "Step-by-step solution restructured using the provided problem-solving template."  
}
```

Example:

```
{  
    "Paraphrase_Question": "A small satellite is launched directly toward a planet and has an initial speed of  $v_{i\_numerical\_value}$  when it is located  $10 R_E_{numerical\_value}$  from the planet's center. Ignoring any atmospheric drag or resistance, determine the satellite's speed  $v_f$  upon reaching the planet's surface.  
The following constants are provided: ( $\text{G}\text{V}$ :  $G_{numerical\_value}$ ,  $\text{M}\text{V}$ :  $M_{numerical\_value}$ ,  
 $R_E\text{V}$ :  $R_E_{numerical\_value}$ ),"  
  
    "Paraphrase_Calculation": "To determine the satellite's speed when it reaches the planet's surface, we follow  
these structured problem-solving steps:\n\n- List Known and Unknowns:\n  - Known: Initial speed =  $v_{i\_numerical\_value}$ , initial distance =  $10 R_E_{numerical\_value}$ ,  
  final distance =  $R_E_{numerical\_value}$ , constants  $G = G_{numerical\_value}$  and  $M = M_{numerical\_value}$ .  
  - Unknown: Final speed  $v_f$ .  
  
- Symbolic Solutions:\n  - Since there are no dissipative forces, total mechanical energy is conserved:  $K_{initial} + U_{initial} = K_{final} + U_{final}$ .  
  - Let  $m$  be the satellite's mass. Kinetic energy is  $(1/2)mv^2$ , and gravitational potential energy is  $-GMm/r$ .  
  - Substituting values:  
     $(1/2)mv_f^2 - GMm/R_E = (1/2)mv_i^2 - GMm/(10R_E)$   
  - Solving algebraically for  $v_f$ :  
     $v_f^2 = v_i^2 + (2GM/R_E)(1 - 1/10)$ .  
  
- Units and Dimensions:\n  - Check that all constants and distances are compatible in terms of dimensions.  
  
- Check Special Cases:\n  - If  $v_{i\_numerical\_value} = 0$ , the equation still works, giving a result based purely on gravitational attraction.  
  - Confirm that the final velocity value is physically reasonable given the context."  
  
Thus, the satellite's final speed  $v_f$  can be computed using the above expression."  
}
```