
SciML Agents: Write the Solver, Not the Solution

Saarth Gaonkar^{*1} Xiang Zheng^{*1} Haocheng Xi¹ Rishabh Tiwari¹

Kurt Keutzer¹ Dmitriy Morozov² Michael W. Mahoney^{1,2,3} Amir Gholami^{1,3}

¹UC Berkeley ²LBNL ³ICSI

Abstract

A large body of recent work in scientific machine learning (SciML) aims to tackle scientific tasks *directly* by predicting target values with neural networks (e.g., physics-informed neural networks, neural ODEs, neural operators, etc.), but attaining high accuracy and robustness has been challenging. We explore an alternative view: use large language models (LLMs) to *write code* that leverages decades of numerical algorithms. We introduce two new datasets: a diagnostic dataset of adversarial “misleading” problems; and a large-scale benchmark of 1,000 diverse ODE tasks. The diagnostic set contains problems whose superficial appearance suggests stiffness, and that require algebraic simplification to demonstrate non-stiffness; and the large-scale benchmark spans stiff and non-stiff ODE regimes. We evaluate open- and closed-source LLM models along two axes: (i) unguided versus guided prompting with domain-specific knowledge; and (ii) off-the-shelf versus fine-tuned variants. Our evaluation measures both executability and numerical validity against reference solutions. We find that with sufficient context and guided prompts, newer instruction-following models achieve high accuracy on both criteria. In many cases, recent open-source systems (e.g., the Qwen3 family) perform strongly without fine-tuning, while older or smaller models still benefit from fine-tuning. Overall, our preliminary results indicate that careful prompting and fine-tuning can yield a specialized LLM agent capable of reliably solving simple ODE problems.

1 Introduction

Recently, several approaches have used machine learning to solve scientific tasks by predicting solutions directly with neural networks. Notable works include: Physics-Informed Neural Networks [1], neural ODEs [2, 3], neural operators [4, 5, 6, 7, 8, 9], Finite-Dimensional operators [10, 11, 12], neural FEMs [13, 14], Fourier Neural Operators [15, 16], and neural discrete equilibria [17] to name a few. Although these approaches have their strengths, training these models to output the solution with high accuracy can be challenging [18].

By contrast, decades of numerical analysis offer robust algorithms for scientific computing problems. This raises a natural question: instead of predicting target values directly, why not use AI models to write code that leverages established numerical solvers?

Prior work has focused on algorithmic correctness and unit tests, but scientific computing introduces additional requirements of numerical suitability tied to problem structure. Currently, no benchmark systematically evaluates the scientific validity of LLM-generated ODE solvers, leaving open how well models reason about mathematical properties to select optimal strategies.

In this work, we address this gap through the following contributions.

- **Robustness Evaluation of Symbolic Reasoning.** We design adversarial, *misleading* ODEs whose superficial form suggests stiffness, but which algebraically simplifies to a non-stiff ODE. The task probes whether models perform genuine symbolic reduction rather than pattern matching, and whether they choose explicit methods when appropriate. We find that while older models struggle more with these examples, newer models demonstrate strong “reasoning” capabilities. In addition, we observe that providing the model with a guided prompt can significantly boost performance.
- **ODE-1000 Benchmark.** To evaluate scientific code generation capability of these models, we create the ODE-1000 benchmark, comprising 1,000 diverse ODE problems. We evaluate the performance by (i) checking if the generated code compiles correctly, and (ii) computing the resulting error of the final numerical solution against the ground truth.
- **Prompting vs. Fine-Tuning.** Using the ODE-1000 dataset, we conduct a detailed empirical analysis across a wide range of open- and closed-source LLMs, considering several scenarios: (i) zero-shot/bare minimum prompt, (ii) guided prompt, and (iii) fine-tuned models. We find that with careful guidance, LLMs can act as practical and reliable SciML agents for a wide range of simple ODE tasks. Newer, larger models are particularly adept at this, often achieving high accuracy with prompting alone, while smaller models still benefit significantly from fine-tuning.

These preliminary results establish initial evidence that careful prompting and targeted fine-tuning can yield practical *SciML coding agents* for simple ODE problems, and that they can provide a foundation for evaluating scientific code generation beyond syntactic correctness.

Due to the page limit, we provide a detailed discussion of related work in Appendix A.

2 SciML Agent Setup

2.1 Task Definition

We focus mainly on the following task: given a natural-language description of a physical or mathematical system governed by an ODE, how can we make an LLM generate a complete and executable Python program that can numerically solve the ODE? A successful LLM output should (of course) return executable code, correct parameters, and a relatively low numerical error. However, the code should also be “scientifically appropriate,” e.g., meaning that the LLM should demonstrate an understanding of the underlying mathematical properties of the ODE to make informed decisions.

2.2 Benchmark datasets

We introduce a benchmark suite composed of two novel datasets. First, we present a specialized Diagnostic Dataset that contains problems that are intentionally “misleading” to probe the LLM’s ability for deeper mathematical reasoning, rather than solely relying on superficial pattern matching. Second, we present a large-scale dataset of 1,000 diverse ODE problems called ODE-1000 that derived from natural-language descriptions, designed to test the end-to-end task of generating complete and numerically accurate solver code. Further details on the construction, examples, and evaluation metrics of these datasets can be found in Appendix B and Appendix C, respectively. The distribution of the data of the ODE-1000 dataset can be found in Appendix F.

2.3 Models

Our evaluation includes a wide array of both **open- and closed-source LLM families**, to ensure broad coverage of the current LLM landscape. Open-source models we tested include the Llama series (Llama2, Llama3, Llama3.1) and the Qwen series (Qwen1, Qwen1.5, Qwen2, Qwen2.5, Qwen3). For a closed-source model, we tested OpenAI’s GPT-4.1. We investigated a wide range of **model sizes**, and distinguished between “older” models (released in 2023–2024) and “newer” instruction-following models (from 2025) to track the evolution of capabilities over successive generations.

2.4 Prompting Methods

While pretrained language models possess strong symbolic capabilities, we observe that these abilities often remain underutilized in zero-shot settings, at least without a proper prompting structure. To explore this, we assess model performance under two different prompting methodologies to understand the effect of contextual guidance. **Base Prompt** only provides only minimal instructions, while **Guided Prompt** is engineered to provide substantial guidance by explicitly instructing the

model to perform step-by-step reasoning and simplify expressions algebraically. Details on these two prompts can be found in Appendix G.

2.5 Fine-Tuning

While newer, larger models often perform well with guided prompting, fine-tuning can be essential for improving the reliability of smaller models. We employ a supervised fine-tuning (SFT) approach to create specialized ODE-solving agents. The training data is derived from our **ODE-1000 benchmark**. We use 900 examples for training and reserve the remaining 100 for testing. For the Qwen models, the fine-tuning data included the intermediate reasoning traces to better leverage their native reasoning capabilities, whereas for the Llama models, only the final Python code was used. We performed **full parameter fine-tuning** for 3 epochs across 8 GPUs.

3 Results

3.1 Symbolic-Equivalence Diagnostic Dataset Evaluation

Older Model Performance on Diagnostic Dataset We first evaluate models released in 2023 and 2024, including the LLaMA series (2, 3, 3.1, and 3.3) and Qwen series (1, 1.5, 2, and 2.5). Results are summarized in Table 1. We observe the following trends.

- **Rapid improvement across LLaMA generations.** The reasoning ability of LLaMA models improves significantly from LLaMA 3 to LLaMA 3.3.
- **Architecture matters more than size.** Newer-generation models often outperform earlier ones, even when smaller in size. For example, the smallest model in a newer generation frequently performs better than the largest model from the previous generation.
- **Scale-performance correlation.** Reasoning ability generally improves with model size.

Newer Model Performance on Diagnostic Dataset We next evaluate models released in 2025, including the Qwen 3 series and GPT-4.1. For both Qwen3-8B and Qwen3-0.6B, we compare their reasoning ability with and without the “thinking mode” (where the model is prompted to reason step by step before producing a final answer). These results are summarized in Table 1, and they lead to the following key observations. The overall trend is visualized in Appendix H.

- **Thinking mode significantly enhances reasoning performance.** The thinking mode leads to notable improvements in accuracy, particularly on the algebraic subset. For instance, the performance of Qwen3-0.6B improves from 37.04% to 75.93% when using thinking mode.
- **Newer models exhibit stronger implicit reasoning.** As shown in Appendix J, there are traces of failed reasoning in which the model incorrectly assumes that a superficially large coefficient directly implies stiffness, without further mathematical derivation. We observed that, compared to earlier models, newer ones tend to perform intermediate reasoning rather than relying solely on pattern matching.

The Role of Prompting on the Diagnostic Dataset We analyze model performance using a *base prompt* and a *guided prompt* (recall Section 2.4). We observe that when using the base prompt, the model exhibits limited performance on the Diagnostic Dataset. As shown in Table 1, by switching to a guided prompt that explicitly encourages step-by-step reasoning and expression simplification, we observe a greatly improved performance and reveal the following observations.

- **Larger models demonstrate stronger instruction-following capabilities.** Within the same model family, we observe that models with more parameters consistently achieve better performance. This suggests that scaling up model size enhances their ability to follow prompts effectively.
- **Mathematical simplification abilities have improved significantly in recent models.** Compared to earlier generations, newer models such as the Qwen3 series and GPT-4.1 achieve over 80% accuracy—even for relatively small models like Qwen3-0.6B. This indicates substantial progress in the models’ capacity for step-by-step reasoning and symbolic manipulation.

Table 1: Accuracy (%) across three mathematical diagnostic datasets (Trigonometric, Algebraic, Inverse Function) under Base vs Guided prompts. Providing guided instructions significantly improves performance, particularly for newer models with stronger instruction-following capabilities.

Model	Llama Models						Qwen Models								
	Trig		Alg		Inv		Model	Trig		Alg		Inv			
	Base	Guided	Base	Alg	Guided	Base		Inv	Guided	Base	Trig	Guided	Base	Alg	Inv
Small Models															
Llama2-7B	1.85	0	0	5.56	1.85	0	Qwen1-7B	4.35	10.87	12.96	5.56	3.64	16.36		
Llama3-8B	6.5	6.5	18.5	31.5	3.6	14.6	Qwen2-7B	0	4.35	11.11	14.81	0	5.45		
Llama3.1-8B	10.9	21.7	27.8	44.4	5.5	31.5	Qwen2.5-7B	28.3	47.8	61.1	77.8	18.2	36.4		
Big Models															
Llama3-70B	2.2	28.3	38.9	87.0	0	25.5	Qwen2.5-72B	26.1	78.3	53.7	83.3	54.5	60.0		
Llama3.3-70B	54.4	82.6	81.5	92.6	74.5	90.9	Qwen3-235B	89.1	97.8	100	100	96.4	100		
Llama3.1-405B	17.4	56.5	50.0	83.3	10.9	67.3	GPT-4.1	21.7	100	92.6	100	76.4	100		
Reasoning Models (Qwen3)															
Qwen3 0.6B	58.7	65.2	37.0	51.9	81.8	80.0	Qwen3 8B	54.4	89.1	77.8	79.6	89.1	98.2		
Qwen3 0.6B [†]	69.6	73.9	75.9	79.6	83.6	80.0	Qwen3 8B [†]	100	100	100	100	100	100		

[†] Guided = reasoning mode activated

Importantly, model failure is *not* simply due to insufficient knowledge or representational capacity, but rather due to the absence of explicit guidance and structured reasoning incentives. As shown in Appendix G, the guided prompt adds an explicit instruction, guiding the model toward multi-step reasoning and suppressing pattern-matching behaviors.

3.2 ODE Dataset Evaluation

We evaluate on ODE-1000 using three metrics: (i) **Code execution rate** — fraction of code that compiles and runs end-to-end; (ii) **Accuracy** — fraction of runnable cases with relative L_2 error < 0.01 ; and (iii) **Mean relative L_2 error** — average error (clamped at 1 to reduce outlier effects).

We observe that fine-tuning produces quantitative improvements for small models. See Table D.1 in Appendix for a summary. For example, for the Qwen-3-0.6B model, the code execution rate jumps from 27% to 97%, the mean relative L_2 drops from 0.3705 to 0.1043, and the accuracy among successful runs increases from 62.96% to 87.63%. Llama2-7B shows a similar improvement, where execution goes from 21% $\rightarrow$ 100% and accuracy 47.62% $\rightarrow$ 87%.

These improvements can be mainly classified into two categories.

- **Fewer syntax errors.** Fine-tuning the models with the ODE-1000 dataset can greatly reduce syntax errors, which is necessary for ODE solvers to produce executable code.
- **Improved numerical validity.** Fine-tuning also greatly improves numerical accuracy by enhancing the model’s ability to correctly derive and set up an equation and choose the correct solvers.

For both of these reasons, our results demonstrate that, for smaller and/or older models, fine-tuning can be an effective way to improve performance. On the other hand, for newer reasoning models that are sufficiently large (such as Qwen3-8B), fine-tuning is often not necessary; instead, careful prompting with in-context examples (evaluation prompt in Appendix E) is enough to achieve near 100% performance.*

4 Conclusion and Future Work

We asked whether LLMs can serve as *SciML agents* that translate natural-language ODE descriptions into runnable, *scientifically appropriate* code. To study this, we introduced two benchmarks: a Symbolic-Equivalence diagnostic suite that probes algebraic simplification and solver choice, and ODE-1000, a large-scale collection of diverse problems. Our evaluation, measuring both *executability* and *numerical validity* against reference solutions, shows that careful guidance substantially

*It is important to note that further fine-tuning may be also required if the Agent has to generate code in completely new domains, on which the base model has not been extensively trained. However, the scope that we considered here involves common ODEs for which pretrained models have some general knowledge.

improves performance: newer instruction-following models often achieve high accuracy with domain-aware prompting, while smaller or older models benefit markedly from fine-tuning. Together, these results indicate a practical path to specialized LLM agents that leverage mature numerical solvers rather than predicting solutions directly, and they establish a foundation for assessing scientific code generation beyond syntactic correctness.

References

- [1] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.
- [2] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31, 2018.
- [3] Amir Gholami, Kurt Keutzer, and George Biros. Anode: Unconditionally accurate memory-efficient gradients for neural odes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 730–736, 2019.
- [4] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George E. Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.
- [5] Kaushik Bhattacharya, Bamdad Hosseini, Nikola B Kovachki, and Andrew M Stuart. Model reduction and neural networks for parametric pdes. *The SMAI journal of computational mathematics*, 7:121–157, 2021.
- [6] Nikola B. Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew M. Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces with applications to pdes. *Journal of Machine Learning Research*, 24(89):1–97, 2023.
- [7] Nicholas H Nelsen and Andrew M Stuart. The random feature model for input-output maps between banach spaces. *SIAM Journal on Scientific Computing*, 43(5):A3212–A3243, 2021.
- [8] Anima Anandkumar, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Nikola Kovachki, Zongyi Li, Burigede Liu, and Andrew Stuart. Neural operator: Graph kernel network for partial differential equations. In *ICLR 2020 workshop on integration of deep neural models and differential equations*, 2020.
- [9] Ravi G Patel, Nathaniel A Trask, Mitchell A Wood, and Eric C Cyr. A physics-informed operator regression framework for extracting data-driven continuum models. *Computer Methods in Applied Mechanics and Engineering*, 373:113500, 2021.
- [10] Jonas Adler and Ozan Öktem. Solving ill-posed inverse problems using iterative deep neural networks. *Inverse Problems*, 33(12):124007, 2017.
- [11] Saakaar Bhatnagar, Yaser Afshar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, 64(2):525–545, 2019.
- [12] Yuehaw Khoo, Jianfeng Lu, and Lexing Ying. Solving parametric pde problems with artificial neural networks. *European Journal of Applied Mathematics*, 32(3):421–435, 2021.
- [13] Bing Yu et al. The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1):1–12, 2018.
- [14] Leah Bar and Nir Sochen. Unsupervised deep learning algorithm for pde-based forward and inverse problems. *arXiv preprint arXiv:1904.05417*, 2019.
- [15] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. In *International Conference on Learning Representations (ICLR)*, 2021.
- [16] Yuwei Fan, Cindy Orozco Bohorquez, and Lexing Ying. Bcr-net: A neural network based on the nonstandard wavelet form. *Journal of Computational Physics*, 384:1–15, 2019.
- [17] Jose Antonio Lara Benitez, Junyi Guo, Kareem Hegazy, Ivan Dokmanic, Michael W. Mahoney, and Maarten V. de Hoop. Neural equilibria for long-term prediction of nonlinear conservation laws. Technical report, arXiv preprint arXiv:2501.06933, 2025.

- [18] Aditi S. Krishnapriyan, Amir Gholami, Shandian Zhe, Robert M. Kirby, and Michael W. Mahoney. Characterizing possible failure modes in physics-informed neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, 2021.
- [19] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In *NeurIPS 2021 Track on Datasets and Benchmarks*, 2021.
- [20] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Łukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [21] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [22] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR, 23–29 Jul 2023.
- [23] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024. Oral.
- [24] Jingquan Wang, Andrew Negrut, Harry Zhang, Khailanii Slaton, Shu Wang, Radu Serban, Jinlong Wu, and Dan Negrut. Chronollm: Customizing language models for physics-based simulation code generation. *arXiv preprint arXiv:2508.13975*, 2025.
- [25] Hongchao Jiang, Yiming Chen, Yushi Cao, Hung-yi Lee, and Robby T. Tan. Codejudgebench: Benchmarking llm-as-a-judge for coding tasks. *arXiv preprint arXiv:2507.10535*, 2025.
- [26] Jiayi Sheng, Luna Lyu, Jikai Jin, Tony Xia, Alex Gu, James Zou, and Pan Lu. Solving inequality proofs with large language models. *arXiv preprint arXiv:2506.07927*, 2025.
- [27] Zenan Li, Zhaoyu Li, Wen Tang, Xian Zhang, Yuan Yao, Xujie Si, Fan Yang, Kaiyu Yang, and Xiaoxing Ma. Proving olympiad inequalities by synergizing llms and symbolic reasoning. In *International Conference on Learning Representations (ICLR)*, 2025.
- [28] Yangqiaoyu Zhou, Haokun Liu, Tejes Srivastava, Hongyuan Mei, and Chenhao Tan. Hypothesis generation with large language models. *arXiv preprint arXiv:2404.04326*, 2024.
- [29] Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Mamo: A mathematical modeling benchmark with solvers. *arXiv preprint arXiv:2405.13144*, 2024.
- [30] Michael Shalyt, Rotem Elimelech, and Ido Kaminer. Asymob: Algebraic symbolic mathematical operations benchmark. *arXiv preprint arXiv:2505.23851*, 2025.
- [31] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 2023.
- [32] Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 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, December 2023. Association for Computational Linguistics.
- [33] Shanda Li, Tanya Marwah, Junhong Shen, Weiwei Sun, Andrej Risteski, Yiming Yang, and Ameet Talwalkar. Codepde: An inference framework for llm-driven pde solver generation. *arXiv preprint arXiv:2505.08783*, 2025.

- [34] Mauricio Soroco, Jialin Song, Mengzhou Xia, Kye Emond, Weiran Sun, and Wuyang Chen. Pde-controller: LLMs for autoformalization and reasoning of PDEs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, Vancouver, Canada, 2025. PMLR. ICML 2025.
- [35] Qile Jiang, Zhiwei Gao, and George Em Karniadakis. Deepseek vs. chatgpt vs. claude: A comparative study for scientific computing and scientific machine learning tasks. *Theoretical and Applied Mechanics Letters*, 15(3):100583, 2025.

A Related Work

LLM based Coding Agents. A variety of benchmarks have been introduced to evaluate LLMs’ coding capabilities. Classic datasets such as APPS [19], HumanEval [20], and MBPP [21] assess function-level code generation through unit tests, while DS-1000 [22] emphasizes data science code synthesis with Python libraries. More recent benchmarks such as SWE-bench [23] capture real-world bug fixing in software repositories, ChronoLLM[24] customize LLMs to generate PyChrono virtual experiments, and CodeJudgeBench [25] investigates the reliability of LLMs as judges for code generation, repair, and test generation.

Coding Agents for Scientific Problems. LLMs have recently been applied to scientific computing, showing strong capabilities in symbolic reasoning and code generation. Benchmarks such as Ineq-Math [26], NeqLIPS [27], HypoGeniC [28], Mamo [29] and ASyMOB [30] underscore the expanding role of LLMs in mathematical problem solving; while systems like PAL [31] and LOGIC-LM [32] combine LLMs with interpreters or symbolic solvers to improve arithmetic and logical reasoning. In the scientific computing domain, frameworks like CodePDE [33] have shown that LLMs can generate executable solvers for PDEs through reasoning, debugging, and test-time scaling, in some cases surpassing both human experts and specialized neural PDE solvers on benchmarks. Another work is PDE-Controller [34] which leverages LLMs to allow control PDE-based tasks through natural language.

Another important work in this direction is [35] which recently benchmarked DeepSeek, ChatGPT, and Claude on a wide range of scientific computing and SciML tasks, including ODEs, PDEs, FEMs, and PINNs. Our work complements theirs: while [35] studies cross-model performance differences of closed-source models, we focus on studying the reasoning capability of open/closed-source models, and explore the impacts of guided prompting vs finetuning of these models on the coding capability of these models on a new set of datasets that we introduce.

B Symbolic-Equivalence Diagnostic Dataset

In numerical ODE integration, *explicit* solvers advance the state directly from the current value. They are simple and efficient, but they can become unstable on stiff systems. In contrast, *implicit* solvers require solving for the unknown future state. They are typically more stable (though costlier), making them preferable for stiff or constrained dynamics — see Appendix I for an empirical study examining how accuracy and runtime vary with equation stiffness. In practice, scientific computing experts first assess whether a problem is stiff, which then strongly influences their solver choice.

Here, we test whether current LLMs can correctly classify ODEs as stiff or non-stiff, an essential step for choosing the *right* algorithm when generating code, and whether they rely on superficial cues (e.g., large coefficients) or perform nontrivial symbolic reasoning.

Dataset Construction. The Diagnostic Dataset comprises three subsets that appear *stiff* before simplification, but that reduce to non-stiff forms: a trigonometric subset, an inverse function subset, and an algebraic subset, described as follows.

1. **Trigonometric subset** uses trigonometric identities, such as $\sin(t + \pi/2) + \cos t = 0$, to simplify the equation.
2. **Inverse-function subset** uses inverse operators collapse, such as $\arcsin(\sin t) = t$ on $[-\pi/2, \pi/2]$, to simplify the equation.
3. **Algebraic subset** uses expansions/factorizations, such as $(x+1)^3 - 6(x^2+1) - (x-1)^3 = 0$, that remove large coefficients after simplification.

In each case, the coefficients appear large, but they can be simplified through mathematical identities. Consequently, the correct choice is always an explicit method. We report accuracy as the proportion of instances for which the model selects an explicit solver. This setup probes basic symbolic reduction capability, rather than more simple pattern matching, by the LLM.

C ODE-1000 Dataset

Overview. The ODE-1000 dataset is a curated dataset consisting of 1,000 ODE problems paired with executable Python solutions. The dataset is designed to evaluate the ability of LLMs to derive a correct ODE and problem setup from a natural-language description, and to produce syntactically valid, numerically stable solver code with an appropriate method choice (e.g., implicit vs. explicit solver). Below, we describe the generation pipeline, example structure, and coverage.

Generation. We synthesize candidate problems with GPT 4.1 (prompt templates in Appendix E) by letting the model (1) derive the ODE from a realistic scenario, where the description is *natural language*, (2) specify an appropriate solver method and the reasoning trace behind the choice, and (3) produce the equation and the initial conditions in SymPy format, and the Python code to solve the equation using the `solve_ivp` function in SciPy library. The prompt is designed to enforce *diversity* across: stiff vs. non-stiff dynamics, orders, time intervals, magnitudes and signs of coefficients, and solver coverage. The target distribution is visualized in Figure F.1 and Appendix F.

Example structure. We present an example of the created dataset for clarity. Each item is stored as JSON with five fields, as follows.

- **description:** A capacitor is initially charged to 10 volts and is discharging through a resistor. The voltage $V(t)$ across the capacitor decreases at a rate proportional to the voltage itself, with a time constant of 2 seconds. Find $V(t)$ for t from 0 to 10 seconds, using 200 evaluation points.
- **sympy_eq:** `Eq(Derivative(y(t), t), -y(t)/2)`
- **initial_condition:** `y(0): 10`
- **reasoning:** This is a classic first-order linear ODE for exponential decay: $dV/dt = -V/2$. The time constant is given as 2 seconds. The equation is not stiff, so RK45 is suitable. The time interval and number of points allow us to observe the voltage decay smoothly.
- **code:**

```
def f(t, V):
    return -V / 2

t_eval = np.linspace(0, 10, 200)
sol = solve_ivp(f, [0, 10], [10], t_eval=t_eval, method='RK45')
```

Verification and filtering. After the dataset creation, for each candidate, we run the provided code in a sandbox to ensure it is syntactically valid, and `solve_ivp` returns a solution on the supplied `t_eval`. Candidates that fail to execute are discarded.

We then obtain $y_{\text{analytical}}(t)$ using SymPy `dsolve` from `sympy_eq` and `initial_condition`. We evaluate both the numerical solution $y_{\text{numerical}}(t_i)$ and the analytical solution on the same grid and compute the relative L_2 error:

$$\text{Rel } L_2 = \frac{\|y_{\text{analytical}} - y_{\text{numerical}}\|_2}{\|y_{\text{analytical}}\|_2}.$$

Examples with $\text{Rel } L_2 > 0.01$ are rejected and filtered out, as they are numerically unstable and thus not suitable for benchmarking. After filtering, this process yields 1000 verified examples that emphasize realistic phrasing and solver sensitivity.

D Fine-tuning Result on ODE-1000 Dataset

This section reports fine-tuning results on the ODE-1000 dataset. Following the setup described in the main paper, we fine-tuned Qwen2-7B, Qwen3-0.6B, Llama-7B, Llama2-7B, and Llama3-8B, with detailed results provided in Table D.1.

Table D.1: Performance of various models on the ODE dataset. We can clearly see that fine-tuning helps, particularly for older or smaller models. However, as we saw in Table 1, newer models can achieve high accuracy without finetuning if given enough context and instructions.

Model	Code Execution Rate	Accuracy	Relative L2 Error
Qwen2-7B	93%	91.40%	0.0841
+ <i>Finetuning</i>	99%	94.95%	0.0459
Qwen3-0.6B	27%	62.96%	0.3705
w/ reasoning	27%	70.37%	0.2913
+ <i>Finetuning</i>	97%	87.63%	0.1043
Llama-7B	N/A	N/A	N/A
+ <i>Finetuning</i>	100%	87%	0.1261
Llama2-7B	21%	47.62%	0.4645
+ <i>Finetuning</i>	100%	87%	0.1169
Llama3-8B	96%	89.58%	0.0734
+ <i>Finetuning</i>	98%	94.9%	0.0516

N/A means the model does not have instruction following ability.

E Prompt Engineering for ODE-1000 Dataset

In this section, we present prompts we adopt to generate and evaluate the ODE-1000 dataset.

ODE Dataset Evaluation Prompt

You are an expert in solving ODEs using Python. Your job is to generate code to solve an ODE problem by making a call to `solve_ivp` with the correct parameters: correct function code, correct method (RK45, RK23, BDF, Radau, LSODA, DOP853, etc.), correct `t_eval`, correct number of evaluation points, etc.

Here's an example problem: Population $P(t)$ grows logistically according to $dP/dt = 0.5 \cdot P \cdot (1 - P/1000)$ with initial population 50. Find $P(t)$ from $t = 0$ to $t = 40$ using 250 evaluation points.

Here's the corresponding example solution: `def f(t, P): return 0.5 * P * (1 - P / 1000) t_eval = np.linspace(0, 40, 250) sol = solve_ivp(f, [0, 40], [50], t_eval=t_eval, method='RK45')`

Now, here's the problem you need to solve: `example["description"]`

In the final output, only return the correct code (function and call to `solve_ivp`), nothing else. Use "sol" to refer to the solution. Make sure that the code is syntactically correct, and do not output any other text.

ODE Data Generation Prompt

Generate batch_size diverse examples of ODEs that can be solved using scipy's solve_ivp. Each example should include:

1. A clear description of the ODE in terms of variables of your choice (you should always give away t_eval and number of evaluation points)
2. The correct corresponding sympy equation
3. The correct initial condition (in valid sympy format)
4. Reasoning to set up/solve the ODE with the correct solve_ivp parameters
5. The correct Python code using solve_ivp with appropriate parameters

The sympy equation and initial conditions should be in terms of y(t) even if the original equation is given in terms of different variables.

The examples should be diverse in terms of:

- Different types of ODEs (different orders, stiff, non-stiff, etc.)
- Different correct usages of numerical methods (BDF, Radau, LSODA for stiff problems and RK45, RK23, DOP853 for non-stiff problems)
- Different time intervals
- Different initial conditions

IMPORTANT:

- Generate real-world problems. The ODE should not be given and needs to be derived from the description. Make it as hard as possible; it should be very challenging to correctly derive the ODE, requiring substantial reasoning.
- In the problem description, do not give away if the ODE is stiff or not stiff.
- Generate an equal number of stiff and non-stiff problems.
- When generating a problem with a given method, make sure that the the problem results in a high error or solver instability if solved using a different method.
- Ensure each example has a known closed-form solution that can be found.

Format the output as a JSON array with each example having only the following fields: description, sympy_eqs, initial_conditions, reasoning, code.

Here is an example:

```
{
  "description": "A population P(t) grows at a rate proportional to its current size with a proportionality constant 0.3. The initial population is 100. Find P(t) from t = 0 to t = 20 using 200 points.",
  "sympy_eq": "Eq(Derivative(y(t), t), 0.3 * y(t))",
  "initial_condition": "y(0): 100",
  "reasoning": "The ODE is directly given: dP/dt = 0.3*P. This is a classic exponential growth equation, non-stiff and first order. RK45 is appropriate. The time domain [0, 20] and 200 evaluation points will capture the exponential curve well. The initial condition is mapped directly.",
  "code": "import numpy as np from scipy.integrate import solve_ivp def f(t, y): return 0.3 * y t_eval = np.linspace(0, 20, 200) sol = solve_ivp(f, [0, 20], [100], t_eval=t_eval, method='RK45')"
```

Make sure the examples are challenging and diverse. Return ONLY the JSON array, with no additional text before or after. It must be a valid JSON array.

F ODE Dataset Statistics

In this section, we present the distribution and statistics for solver methods and ODE orders for the ODE-1000 dataset.

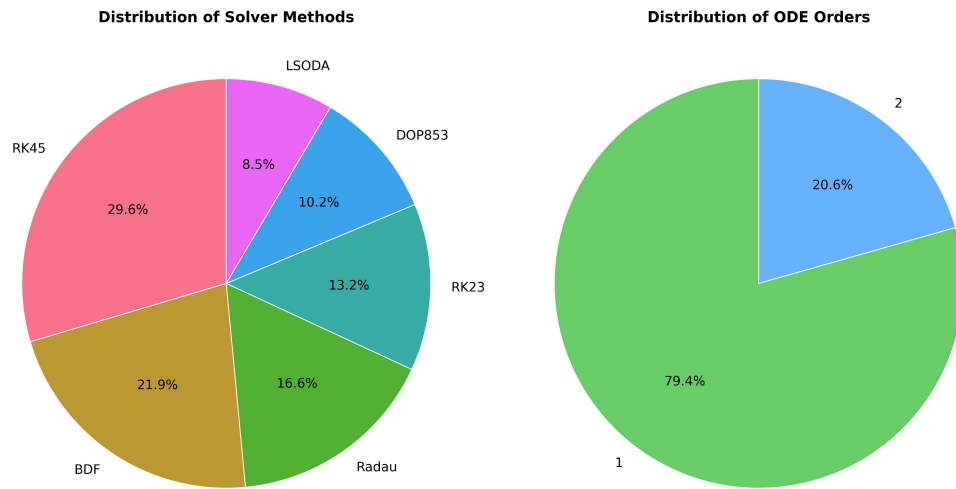


Figure F.1: ODE-1000 dataset distribution of solver methods and ODE orders (1st/2nd).

G Prompt Engineering for Diagnostic Dataset

In this section, we discuss prompts for the Diagnostic Dataset.

The guided prompt introduces two key forms of inductive structure:

- It embeds domain-specific heuristics for stiffness (e.g., "eigenvalues $\ll -1000$ imply stiffness"), giving the model actionable thresholds for decision-making.
- It enforces algebraic and symbolic simplification of the coefficients.

These differences are highlighted in red.

Base System Prompt

You are an expert in numerical methods for solving Ordinary Differential Equations (ODEs). Your task is to determine whether an explicit or implicit solver should be used for the given ODE. Consider the following criteria:

- Explicit solvers (like RK45, RK23, DOP853) are suitable for non-stiff ODEs
- Implicit solvers (like BDF, Radau) are suitable for stiff ODEs
- LSODA automatically switches between stiff and non-stiff solvers, making it ideal for ODEs with varying stiffness.

You will be given an ODE `ode_example` and corresponding `time_interval`.

Please analyze the ODE step by step:

1. First, provide your detailed reasoning about why this ODE is stiff or non-stiff.
2. Then, provide your final answer as either "explicit solver" or "implicit solver". Don't mention anything else, just the solver choice.

Please format your response in json format as follows. IT MUST BE VALID JSON:

```
{
  "reason": "Your detailed analysis here",
  "answer": "explicit solver/implicit solver"
}
```

Guiding System Prompt

You are an expert in numerical methods for solving Ordinary Differential Equations (ODEs). Your task is to determine whether an explicit or implicit solver should be used for the given ODE. Consider the following criteria:

- Explicit solvers (like RK45, RK23, DOP853) are suitable for non-stiff ODEs
- Implicit solvers (like BDF, Radau) are suitable for stiff ODEs
- LSODA automatically switches between stiff and non-stiff solvers, making it ideal for ODEs with varying stiffness.
- Stiff ODEs typically have large negative eigenvalues (much less than -1000)- Non-stiff ODEs have eigenvalues greater than -100

You will be given an ODE `ode_example` and corresponding `time_interval`.

Please analyze the ODE step by step:

1. Given that the ODE is in the form of $dy/dt = p(t)y + q(t)$, please first try to simplify the $p(t)$ and $q(t)$ to make it easier to solve.
2. Then, provide your detailed reasoning about why this ODE is stiff or non-stiff.
3. Finally, provide your final answer as either "explicit solver" or "implicit solver". Don't mention anything else, just the solver choice.

Please format your response in json format as follows. IT MUST BE VALID JSON:

```
{  
  "reason": "Your detailed analysis here",  
  "answer": "explicit solver/implicit solver"  
}
```

H Model Evaluation on Diagnostic Dataset

In this section, we visualize the full result on the diagnostic dataset.

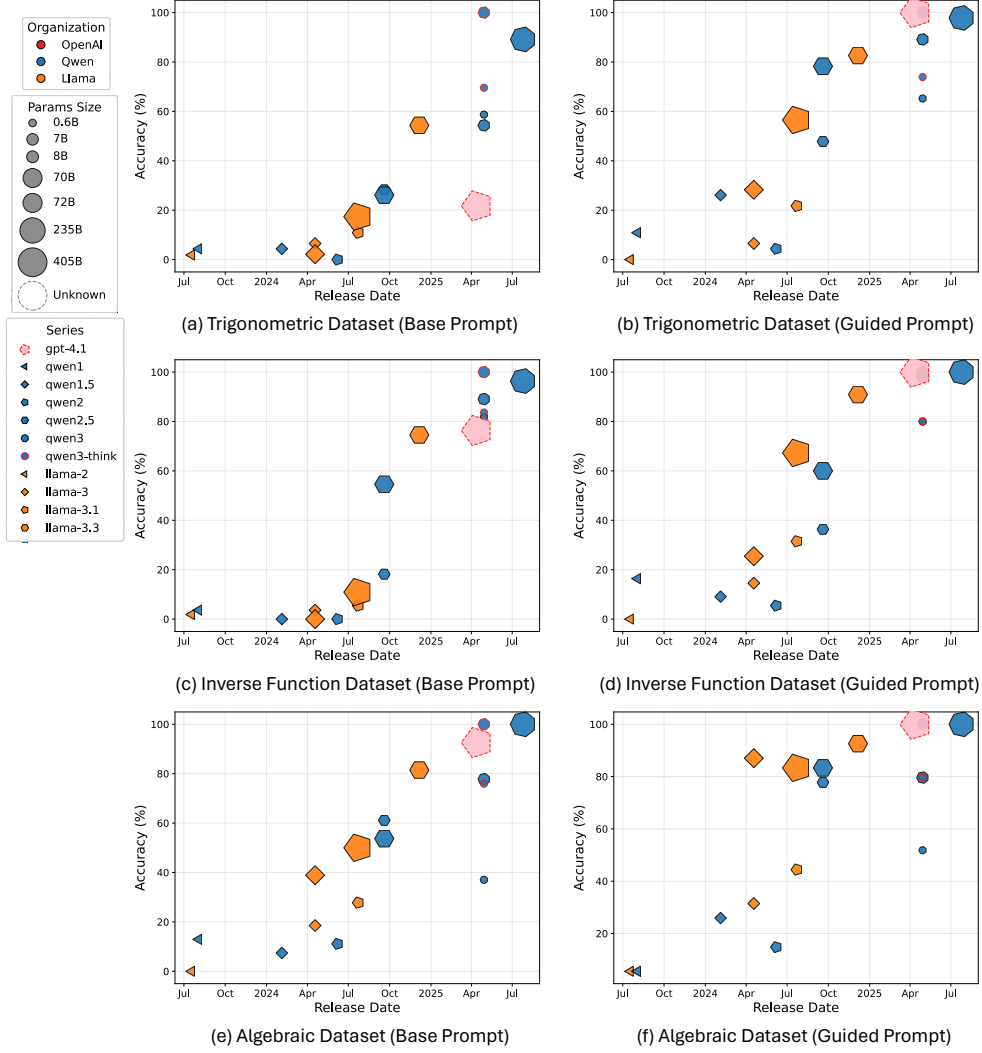
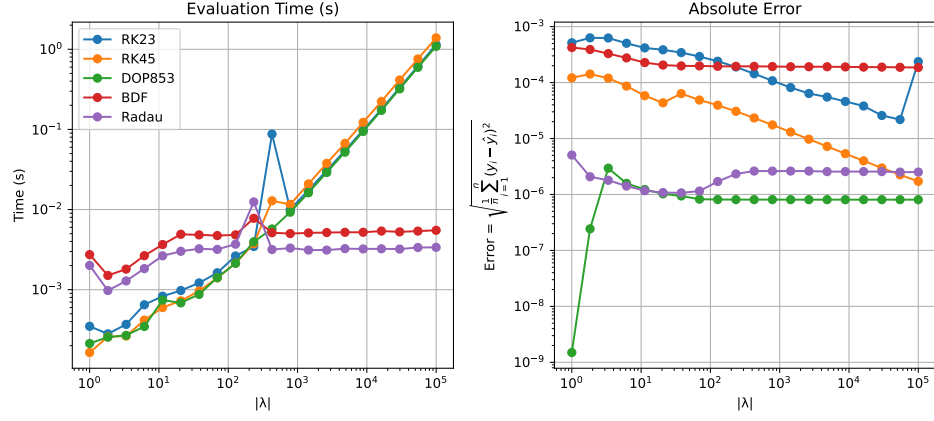


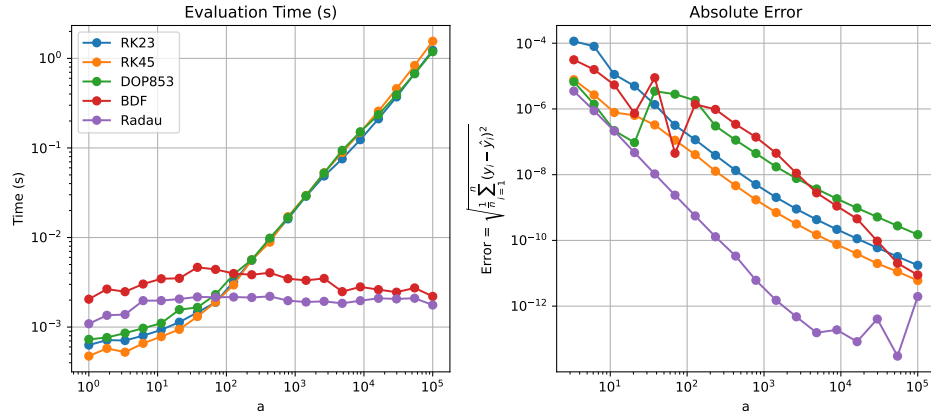
Figure H.1: Evaluation of LLMs across varying release dates and model sizes. (a), (c) and (e) present results on the trigonometric, inverse function and algebraic tasks using the base prompt, while (b), (d) and (f) present results on the same tasks using the guided prompt. We can see a clear trend where newer models are able to achieve higher accuracy. Furthermore, enabling thinking/reasoning mode can boost the accuracy by up to 25%.

I Empirical Study on Explicit/Implicit Solver

In this section, we present additional results on explicit versus implicit solvers.



(a) First-order ODE Experiemnt.



(b) Second-order ODE Experiemnt.

Figure I.1: Empirical comparison of explicit and implicit solvers under varying stiffness. (a) Solver performance on the first-order ODE $\frac{dy}{dt} = \lambda y$: left—runtime; right—RMSE. (b) Solver performance on the second-order ODE $\frac{d^2y}{dt^2} + a \frac{dy}{dt} + y = 0$: left—runtime; right—RMSE.

J Failure Cases from Pattern Matching

In this section, we present reasoning traces showing how GPT-4.1 falls into pattern matching when determining the stiffness of ODEs across the trigonometric, inverse function, and algebraic subsets. We also include some successful cases in which large language models perform the correct mathematical reasoning and arrive at the correct result.

J.1 Trigonometric Subset

1. Failed Case

- ODE example: $dy/dt = (50000 * (\cos(2*t) - 2 * \cos(t)**2 + 1)) * y + \sin(2*t)$,
- Traces of Failed Reasoning: This ODE has the derivative term dy/dt involving a large coefficient (50000) multiplied by y and trigonometric functions of t . Such a large coefficient creates rapid changes in y with respect to t and leads to the presence of both fast and slow dynamics. This multiscale behavior is characteristic of stiff ODEs, where explicit methods would require extremely small timesteps for stability, making them inefficient. Therefore, an implicit solver is more appropriate.

2. Successful Case

- ODE example: $dy/dt = (20000 * (\sin(t)**2 + \cos(t)**2) - 20000) * y + \cos(t)$,
- Traces of Successful Reasoning: The ODE simplifies $(\sin(t))^2 + \cos(t)^2 = 1$, so $dy/dt = (20000 * 1 - 20000) * y + \cos(t) = \cos(t)$. This reduces to $dy/dt = \cos(t)$, a very simple, non-stiff ODE with no dependence on the large coefficient (since it cancels out). There are no rapidly decaying or strongly coupled components in the equation; the solution is smooth and varies on the scale of $\cos(t)$, not 20000. Therefore, stiffness is not an issue.

J.2 Inverse Function Subset

1. Failed Case

- ODE example: $dy/dt = (4e7 * (\sin(\arcsin(t/2)) - t/2)) * y + \exp(t)$,
- Domain: $t \in [0, 1]$,
- Traces of Failed Reasoning: The ODE's coefficient of y contains a large constant factor ($4e7$) that multiplies a function of t , which can lead to very rapid changes in the solution (potentially large negative or positive eigenvalues in the Jacobian). This is characteristic of stiff ODEs, as explicit methods may require prohibitively small step sizes for stability over even a short interval. Thus, an implicit solver is appropriate.

2. Successful Case

- ODE example: $dy/dt = (1e7 * \log(\exp(t + 1)) - 1e7 * (t + 1)) * y + \cos(t)$,
- Domain: $t \in [0, 1]$,
- Traces of Successful Reasoning: The $\log(\exp(t + 1))$ term equals $t + 1$, so $1e7 * \log(\exp(t + 1)) - 1e7 * (t + 1)$ cancels to 0. The Jacobian with respect to y is 0, so there are no stiff eigenvalues and the system is not stiff. Only the $\cos(t)$ term remains, thus an explicit solver is appropriate.

J.3 Algebraic Subset

1. Failed Case

- ODE example: $dy/dt = ((t-5)**4 - t**4 + 20*t**3 - 150*t**2 + 500*t - 625)*y + \sqrt{t+3}$,
- Traces of Failed Reasoning: To determine stiffness, let's analyze the ODE $dy/dt = ((t-5)^4 - t^4 + 20*t^3 - 150*t^2 + 500*t - 625) * y + \sqrt{t+3}$. The right side shows linear dependence on y with a time-varying coefficient $P(t) = (t-5)^4 - t^4 + 20t^3 - 150t^2 + 500t - 625$. We should check for stiffness by looking for rapidly changing or large negative real parts in the eigenvalue (which, for linear scalar ODEs, is just $P(t)$). Evaluating $P(t)$ at the endpoints: $P(0) = 625$, $P(5) = (-625) + (2500) - (3750) + (2500) - (750) - (625) = \text{large negative value}$. So, $P(t)$ varies from positive to large negative within a short interval, potentially becoming very large in magnitude. Such large coefficients can make the ODE stiff, as the solution can rapidly decay or grow depending on sign and magnitude, and explicit solvers may need exceedingly small timesteps for stability.

2. Successful Case

- ODE example: $dy/dt = ((t+5)**3 - t**3 - 15*t**2 - 75*t - 125)*y + \sqrt{t+2}$,
- Traces of Successful Reasoning: To assess stiffness, examine the ODE's right-hand side: $dy/dt = ((t+5)^3 - t^3 - 15t^2 - 75t - 125) * y + \sqrt{t+2}$. The coefficient of y simplifies as follows: $(t+5)^3 - t^3 - 15t^2 - 75t - 125 = (t^3 + 15t^2 + 75t + 125) - t^3 - 15t^2 - 75t - 125 = 0$. Therefore, $dy/dt = \sqrt{t+2}$. This is a simple function of t , and there is no y dependency beyond the initial value; thus, the ODE is not stiff. Its explicit integration is straightforward and does not require an implicit solver.