Fine-Tuning Phi-3 for Asymptote Code Generation from Mathematical Diagrams

Shivani Kedila

1. Introduction

This project explores the use of parameter-efficient fine-tuning techniques to adapt a small language model (Phi-3 Mini, 3.8B) for the task of translating mathematical diagram prompts into compilable Asymptote code. The broader context involves applying generative AI in K–12 geometry education to assist in vector diagram generation. A curated dataset was constructed from open-source Asymptote repositories, filtered and augmented via parametrization. A QLoRA-based pipeline was implemented under Colab hardware constraints, leveraging 4-bit quantization and LoRA adapters to enable instruction fine-tuning within limited memory environments.

2. Methodology

2.1. Model Selection

Microsoft's Phi-3 Mini (4k context variant) was selected as the base model due to both computational and methodological considerations. Initially, larger models such as LLaMA or Mixtral were considered, but memory and compute constraints in the Colab Pro environment made them impractical to fine-tune effectively. Phi-3 Mini's relatively small 3.8B parameter size, combined with its compatibility with 4-bit QLoRA techniques, allowed it to be trained within a constrained 40 GB GPU environment. Beyond practicality, recent work such as the LIMA study (Zhou et al., 2023) have shown that, when fine-tuned on high-quality datasets, smaller, instruction-tuned models can outperform larger models on alignment tasks. Phi-3's instruction-tuned pretraining and 4k token context window made it a good candidate for the task of translating prompt-style inputs into compilable Asymptote code.

2.2. Dataset Curation

The dataset was cloned from the asymptote-exemples GitHub repository, containing over 500 categorized .asy files with associated .tag and .id metadata. A filtering pass, removing non-relevant samples, retained 424 samples as relevant to K–12 curriculum. Several custom Python scripts were developed to parse, normalize, and augment these samples through parameter substitution and controlled variable injection. This expansion yielded a dataset of ~2000 samples, from which 1727 valid entries were retained after png-rendering validation. Descriptions were then generated for each example using GPT-4o via OpenAI's API, (constrained to 200 tokens per prompt). The data was then put through a randomized 80/5/15 training/validation/test split was used, with a smaller validation set due to repeated GPU memory overflows during early experiments.

2.3. Training Procedure

The model was fine-tuned using 4-bit quantization (via BitsAndBytesConfig) and Low-Rank Adaptation (LoRA) through the peft library. The base Phi-3 model was loaded using transformers with eager attention, and LoRA adapters were injected on common transformer projection modules (q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj) with a rank of 16 and a=32. Tokenization was done using Phi-3's default tokenizer with right padding and a 1024 token maximum length. The loss function was cross-entropy with label

shifting, implemented natively by HuggingFace's Trainer. Logging and evaluation occurred every epoch, with early stopping and best model saving enabled. This model training was executed on a Colab Pro A100 instance with 40 GB VRAM. Training was found to minimize token-level cross-entropy loss, which is a standard objective for causal language modeling, using label shifting as implemented by HF's Trainer.

Figure 1. Training Hyperparameters

Parameter	Value
Base model	Phi-3 Mini (3.8B)
Dropout	0.05
Batch size	2 (acc. 16 → 32)
Batch size	2 (acc. 16 → 32)
Precision	bf16
Quantization	4-bit (BitsAndBytes)
Optimizer	AdamW (default)

2.4. Evaluation

Evaluation was performed on a held-out test set (15%) using three metrics:

- **Syntactic Validity**: Measured via compilation success using the asy command-line tool.
- **Syntactic Similarity**: Compared generated .asy files to ground-truth using SequenceMatcher and exact match.
- **Visual Fidelity**: Intended but not completed due to non-compiling asymptote code.

The evaluation pipeline consisted of three modular Python scripts:

1. Code generation from test prompts
2. Syntax evaluation (compilation and logging)
3. Similarity comparison via string metrics

3. Experimental Results

Due to GPU memory constraints on Colab Pro (A100, 40 GB VRAM), early stopping was disabled and the number of training epochs was limited to 5. The training process still demonstrated steady improvements in loss over the epochs:

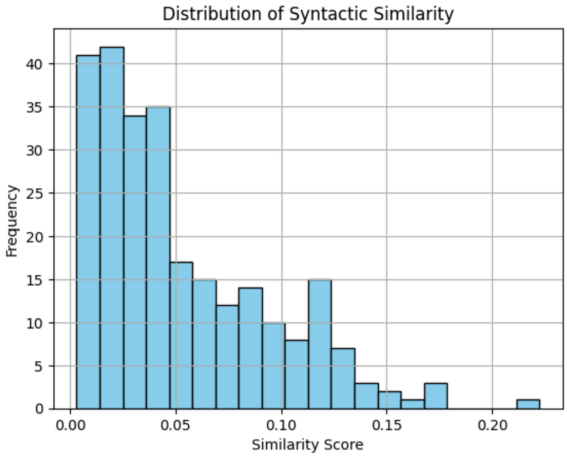
Figure 2. Train/Val Loss Values

Epoch	Training Loss	Validation Loss
1	—	1.360
2	1.450	1.194
3	1.209	1.082
4	0.998	1.001
5	0.840	1.015 (overfitting begins)

Test evaluation yielded the following:

- **Exact Match Rate**: 0.00%
- **Average String Similarity**: 5.3%
- **Syntactic Compilation Success**: 0/260

Figure 3. Similarity Bar Plot



Despite extremely low quantitative metrics, qualitative inspection (Section 4.4) showed partial structure retention in some completions, particularly for early geometric primitives.

4. Insights and Challenges

4.1 Memory and Truncation

Colab GPU memory limits allowed for experimentation with QLoRA and 4-bit quantization. However, despite being effective for training feasibility, these constraints limited sequence length, occasionally truncating completions. This was evident in generation outputs abruptly ending mid-line, suggesting that max_new_tokens should be increased post-training.

4.2 Instruction Bias and Import Omissions

Many generated samples omitted critical packages (e.g., import geometry;, size(…)) and misused *point* in place of *pair*, likely due to training descriptions dominating the prompt format. This suggests that few-shot or better-conditioned prompts could improve syntactic consistency.

4.3 Dataset Augmentation Tradeoffs

The parameterization strategy increased dataset size, but introduced occasional noise (e.g., inconsistent spacing, missing closure). These inconsistencies may have compounded fine-tuning difficulties by introducing malformed code patterns.

4.4 Qualitative Evaluation

Although the model failed to generate fully compilable Asymptote code in all test cases, qualitative inspection of several outputs revealed partial structural alignment with the intended geometric content. One case study is provided below.

Sample: ID 0542.asy

Ground Truth Code:

```
size(8cm,0);
path p1=(0,0)--(1,1);
path p2=(2,0)--(3,1);
draw(p1,linewidth(5mm));
draw(p2,linewidth(5mm));
draw(p1--p2, 3mm+.8blue);
draw(reverse(p1)--p2, 1mm+.8red);
```

Generated Code (truncated):

```
import geometry;

size(10cm,0cm);

// Draw the first segment
draw(A--B, thick(5mm), black);

// Draw the second segment
draw(C--D, thick(5mm), black);

// Connect the segments with a blue line
draw(B--C, thick(3mm), blue);
```

Observations:

Correct High-Level Structure: The model successfully inferred the presence of two segments and an intermediate connector.

Style Preservation: The thickness (5mm) and basic coloring (black, blue) were preserved, suggesting that the model internalized some graphical conventions from the training data.

Failure to Compile: The generated code is missing definitions for variables such as A, B, C, and D, and misuses the draw function without constructing proper pair or path objects. This resulted in syntactic failure.

Loss of Detail: The original code includes nuanced styling such as reversed paths and lower opacity (.8red), which were not captured in the generated output. This hints at limitations in the model's ability to handle fine-grained style directives.

Prompt Contamination: The inclusion of the natural-language prompt itself in the generated code (truncated for this case study) indicates improper decoding boundaries and possibly excessive retention of system/user instructions.

5. Future Improvements

To address the current limitations observed during training and evaluation, several future directions are proposed. These improvements aim to enhance both syntactic correctness and overall alignment between generated code and intended geometric structure.

5.1. Post-Processing for Syntax Repair

A lightweight rule-based post processor could be developed to correct common failure modes, including:

- Automatically adding necessary preambles (import geometry;, unitsize(...), etc.)
- Replacing incorrect primitives such as point with pair
- Detecting and closing incomplete statements (e.g., unterminated draw(...) calls). This system could significantly boost compilation rates without requiring model retraining, functioning as a syntactic filter on top of generation.

5.2. Prompt Engineering and Template Optimization

Prompt structure greatly affects generation quality. Future iterations could:

- Include few-shot examples to reinforce correct code formatting
- Avoid prompt contamination by separating system/user content more cleanly
- Adopt stricter system-level instructions to enforce use of specific Asymptote modules and syntax patterns

5.3. Sequence Length and Token Budget

Several generations were truncated, suggesting that the 1024-token limit was insufficient. Future fine-tuning efforts could:

- Train with max_seq_length=2048 to accommodate more complex diagrams
- Increase max_new_tokens during inference to allow more complete completions
- Use a sliding-window or continuation generation approach for long diagrams

5.4. Dataset Improvements

While parametrization increased dataset diversity, it also introduced noise. Some improvements that can be made include::

- Filtering out degenerate templates (e.g., overlapping points, malformed loops)
- Introducing structured metadata (e.g., number of objects, style tags) to support conditional generation

5.5. Evaluation Enhancements

Current evaluation was limited to syntactic and textual similarity. More rigorous evaluation should include:

- **Visual Fidelity**: Using SSIM or perceptual hash to compare generated images to reference images
- **Functionality Scoring**: Binary compile success is limited—soft scoring (e.g., number of successful draw(...) calls) may yield more insight
- **Human Evaluation**: Asking geometry educators to assess code clarity and usefulness for instructional purposes

5.6. Model Architecture and Fine-Tuning Strategies

Further gains could be achieved by experimenting with:

- LoRA rank tuning and dropout scheduling
- Curriculum learning (starting with simpler figures and gradually increasing complexity)
- Exploring other compact instruction-tuned models (e.g., Gemma, Zephyr) to assess architecture-specific strengths (with access to higher-performance computing)

Despite falling short on strict syntactic correctness, the model demonstrates emerging diagrammatic reasoning. Its partial reproduction of geometric structures indicates potential for future development. With improved prompts, syntax repair, and architectural refinement, this foundation could evolve into a viable tool for AI-assisted geometry education.

6. References

Aftandilian, E., Bao, Q., Sundararajan, V., & Mockus, A. (2024). *Exploring parameter-efficient fine-tuning techniques for code generation with large language models*. *ACM Transactions on Software Engineering and Methodology, 33*(2), 1–32. https://doi.org/10.1145/3714461

DataCamp. (2024). *Top 15 small language models for 2025*. https://www.datacamp.com/blog/top-small-language-models

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., & Jiang, D. (2020). *CodeBERT: A pre-trained model for programming and natural languages* (arXiv:2009.07118). arXiv. https://doi.org/10.48550/arXiv.2009.07118

Napalkova, L. (2023, August 7). *Fine-tuning small language models: Practical recommendations*. *Medium*. https://medium.com/@liana.napalkova/fine-tuning-small-language-models-practical-recommendations-68f32b0535ca

Pivaldi. (n.d.). *asymptote-exemples* [GitHub repository]. GitHub. https://github.com/pivaldi/asymptote-exemples

Zhou, C., Sharma, A., Koratana, A., Xu, J., Chang, K.-W., & Raffel, C. (2023). *LIMA: Less Is More for Alignment* (arXiv:2305.11206). arXiv. https://doi.org/10.48550/arXiv.2305.112