university of groningen

faculty of science and engineering

Credit Assignment for Surrogate Gradient Learning Rules in Spiking Neural Networks

Manvi Agarwal

university of groningen
faculty of science and engineering

University of Groningen

Automating Molecular Dynamics Simulations via a Large Language Model Agent

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen

under the supervision of
Prof. dr. Andrea Giuntoli (Zernike Institute for Advanced Materials, University of Groningen)
and
Prof. dr. Matthia Sabatelli (Artificial Intelligence, University of Groningen)

Junsheng Yin (s4774280)

May 27, 2025

Contents

Acknowledgments

Abstract

1 Introduction

In recent years, the development of Large Language Models (LLMs) such as GPT-3 [1] and GPT-4 has sparked a revolution in natural language processing and generation tasks. These models, trained on massive corpora, exhibit strong capabilities in not only textual understanding but also code synthesis, including generation, completion, and correction. As such, LLMs have become increasingly valuable tools in both industrial and academic settings for assisting software development, education, and scientific computing.

In the domain of computational physics and chemistry, the automation of simulation workflows remains a challenging yet highly impactful goal. One prominent simulation tool is LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [2], a classical molecular dynamics (MD) package used for modeling atoms and particles over time. Despite its power and versatility, LAMMPS requires users to construct input scripts using a domain-specific language (DSL), which demands precision and in-depth knowledge of simulation parameters, force fields, and physical modeling principles.

Generating correct and effective LAMMPS input files is often a significant barrier for new users and researchers working across disciplinary boundaries. The scripts typically involve a set of interdependent commands specifying atomic configurations, interatomic potentials, simulation boundaries, thermodynamic controls, and output protocols. An error in one line can cause the simulation to crash or produce unphysical results. As such, manual creation and debugging of these files can be both time-consuming and error-prone.

The potential for LLMs to assist in this process is considerable. By translating natural language descriptions into executable LAMMPS input scripts, LLMs could significantly lower the entry barrier for performing MD simulations. However, the inherent complexity and domain specificity of LAMMPS syntax pose unique challenges. General-purpose LLMs are not explicitly trained on DSLs like LAMMPS and may therefore produce incomplete, incorrect, or unexecutable scripts.

Previous research on generative AI for domain-specific input files, such as White and McDaniel's work on using LLMs to generate ORCA input files for quantum chemistry [3], has shown the promise of adapting LLMs for scientific use cases. However, LAMMPS introduces additional challenges due to the structural depth and modularity of its input scripts. It is not enough to simply learn the syntax; successful generation also depends on understanding the physics encoded in the script's design.

This master thesis aims to systematically explore and evaluate three complementary strategies for improving LLM performance in LAMMPS input generation:

- **Prompt Engineering:** Crafting optimized prompt templates to improve output fidelity through zero-shot and few-shot learning, as well as advanced prompting methods like Chain-of-Thought (CoT) and Tree-of-Thought (ToT).
- **Retrieval-Augmented Generation (RAG):** Supplementing model inputs with contextual information retrieved from relevant documents and repositories, including the LAMMPS manual, community tutorials, GitHub repositories, and domain-specific literature.
- **Fine-Tuning:** Training LLMs on a corpus of LAMMPS input files to instill domain knowledge and improve generation quality on both seen and unseen tasks.

These methods are not mutually exclusive and may in fact be complementary. Accordingly, this thesis also explores combinations of these strategies (e.g., Fine-Tuning + RAG) to investigate possible synergy or redundancy between them.

To evaluate these approaches, we build a curated benchmark consisting of representative LAMMPS tasks drawn from official examples, open-source repositories, and brute-force synthetic combinations of commands. We measure performance based on several criteria: syntactic correctness (does the file parse?), semantic validity (are the physical parameters coherent?), and functional correctness (does the script run and produce stable simulations?).

Research Objectives and Questions

The overarching objective of this thesis is to explore how LLMs can be optimized to automatically generate high-quality LAMMPS input scripts from natural language prompts. This leads to the following research questions:

1. How well do general-purpose LLMs perform at generating LAMMPS input files out-of-the-box?
2. Can prompt engineering alone significantly improve the syntactic and semantic quality of LAMMPS scripts?
3. Does retrieval-augmented generation offer measurable gains over baseline and prompt-engineered models by leveraging external documentation?
4. Can fine-tuned models better capture the intricacies of LAMMPS input logic, and do they outperform prompt engineering and RAG?
5. Are combinations of fine-tuning, RAG, and CoT prompting more effective than individual methods?

Answering these questions will not only shed light on the feasibility of automating LAMMPS scripting but also contribute general insights into the design of LLM-based tools for domain-specific code generation.

Thesis Structure

The rest of this thesis is organized as follows:

- **Chapter 2** reviews relevant literature on LLM-based code generation, scientific computing applications, and existing tools in molecular simulation.
- **Chapter 3** describes the methodological framework, including dataset preparation, experimental design, and implementation of each optimization strategy.
- **Chapter 4** presents and analyzes the experimental results, comparing performance across strategies and combination methods.
- **Chapter 5** discusses the findings, implications for scientific practice, and future research opportunities.
- **Chapter 6** concludes the thesis with a summary of contributions and recommendations for deploying AI-assisted simulation tools.

2 Related Work

2.1 Large Language Models

Large Language Models (LLMs) have transformed the landscape of artificial intelligence, offering unprecedented performance in a wide range of natural language processing (NLP) and understanding tasks. These models are typically based on the Transformer architecture introduced by Vaswani et al. [4], which relies on self-attention mechanisms to model relationships between words in a sequence, regardless of their distance from each other. This architecture laid the foundation for a series of increasingly capable models that scale up both in terms of data and parameters.

Among the first influential transformer-based models were BERT (Bidirectional Encoder Representations from Transformers) [5] and GPT-2, which demonstrated how pretrained models could be fine-tuned or used directly for a wide range of NLP tasks. BERT, in particular, introduced a bidirectional training method that allowed it to deeply understand context, while GPT-2 and its successors leveraged autoregressive training to generate coherent and contextually appropriate text.

The advent of GPT-3 [1], a model with 175 billion parameters, marked a turning point for LLMs. GPT-3 demonstrated strong few-shot and even zero-shot capabilities, meaning it could perform tasks with little or no task-specific training. It became evident that model scale, combined with vast training corpora, allowed these systems to generalize across diverse tasks. Soon after, models like Codex [6], a descendant of GPT-3, were fine-tuned on code-specific datasets, making them significantly more effective at programming tasks compared to their general-purpose counterparts. Codex powers tools like GitHub Copilot and has shown strong performance in translating natural language queries into functional code.

Additional notable developments include Google's PaLM [7], which pushed model scaling even further (up to 540 billion parameters) and achieved state-of-the-art performance across multiple benchmarks. Meta AI's LLaMA [8] emphasized training efficiency and openness, making powerful language models more accessible to the research community. Each of these models benefited from massive datasets—ranging from books and websites to source code repositories—and contributed to the growing capabilities of LLMs.

Despite their success, LLMs face notable challenges when applied to domain-specific languages (DSLs) like the one used in LAMMPS. DSLs are typically underrepresented in training corpora and exhibit complex syntax and semantics, which differ significantly from the natural language and general-purpose programming languages that dominate LLM training datasets. Furthermore, DSLs often require context-sensitive logic and expert domain knowledge to use effectively.

These limitations have led to an increased interest in strategies for adapting LLMs to specialized domains. Common techniques include prompt engineering, retrieval-augmented generation (RAG), and model fine-tuning on curated domain-specific corpora. These strategies aim to enhance the relevance and correctness of model outputs by either guiding the model more effectively or expanding its internal knowledge base.

Moreover, scientific and engineering applications present additional constraints: correctness, reproducibility, and safety are paramount. Issues such as hallucination—where the model generates plausible but false information—and brittleness to prompt variations must be addressed before LLMs can be safely deployed in high-stakes domains. Understanding and mitigating these risks is a key part of current research into the reliable use of LLMs in scientific workflows.

2.2 LLM Applications to Molecular Dynamics

The application of Large Language Models (LLMs) in molecular dynamics (MD) is a burgeoning field with transformative potential. LLMs offer opportunities to simplify, automate, and enhance various stages of the MD simulation pipeline. From generating input scripts to analyzing simulation outputs, LLMs can function as intelligent assistants, reducing human error, improving productivity, and enabling broader accessibility for non-specialists.

One promising area is the automatic generation of MD simulation input scripts. LLMs can translate high-level natural language prompts into structured input formats required by engines like LAMMPS or GROMACS. For instance, a user might describe a desired simulation scenario—"simulate a copper nanowire under tensile strain at room temperature using EAM potential"—and receive a corresponding '.in' script for LAMMPS. This task involves code synthesis, logic modeling, and contextual reasoning, areas where recent LLMs have demonstrated impressive potential [6, 9].

Another key function involves selecting appropriate interatomic potentials or force fields based on the described system and simulation objectives. For example, an LLM might suggest the TIP4P/2005 water model for accurate thermodynamic property prediction and justify this choice based on simulation constraints. This kind of intelligent recommendation mimics domain-expert behavior and can be enabled using few-shot prompting or retrieval-based reasoning [10].

LLMs can also support users in setting up simulation systems, such as generating lattices, defining regions, assigning boundary conditions, or initializing velocities. With context-aware prompting, the model can produce relevant LAMMPS commands like lattice, region, or create_atoms, customized to specific materials and simulation goals. This functionality benefits both experienced users looking to accelerate script writing and beginners learning how to construct a valid simulation.

The utility of LLMs extends beyond input generation. Simulation outputs—such as 'log.lammps' files or trajectory dumps—are often difficult to interpret. LLMs could help analyze energy profiles, identify anomalies in temperature trends, or suggest improvements based on convergence patterns. They might also offer visualization scripts or natural language interpretations of structural outputs like radial distribution functions. These capabilities, particularly when paired with postprocessing pipelines, could greatly improve user experience and insight generation [11].

In more advanced scenarios, LLMs can serve as iterative design assistants. After reviewing the results of a simulation, the model might recommend follow-up changes such as modifying thermostat parameters, reducing time step sizes, or switching to alternative potentials. When combined with structured feedback loops or rule-based systems, this opens the door to adaptive simulation workflows driven by intelligent agents [11, 10].

Finally, LLMs hold strong potential as educational tools. They can answer user queries about MD theory, command syntax, or simulation design principles. By interacting with a conversational assistant, users might ask questions like "What does 'fix nvt' do?" or "Why should I not use a large timestep?"—and receive accessible, accurate responses. In academic environments, this can enhance learning and help students build confidence with simulation software [10].

Despite the wide applicability of LLMs, few studies have directly addressed the automation of LAMMPS workflows using these models. Most existing tools rely on graphical interfaces or scripting wrappers such as ASE, which abstract away some complexity but lack the adaptability and accessibility of natural language interaction. This thesis investigates how LLMs can fill that gap by serving as intelligent, context-aware generators of LAMMPS scripts. By benchmarking strategies such as prompt engineering, retrieval-augmented generation (RAG), and fine-tuning, this research aims to identify effective approaches for improving LLM performance on LAMMPS scripting tasks.

2.3 Data Augmentation through Synthetic Code Generation

In domains where labeled training data is limited or costly to collect—such as domain-specific languages (DSLs) like those used in scientific computing—synthetic data generation has emerged as a valuable strategy for augmenting code synthesis tasks. Synthetic datasets can be programmatically generated by combining grammar rules, templates, and logical constraints to simulate realistic, diverse, and structurally valid examples. This is especially useful for bootstrapping learning in scenarios where real-world code samples are sparse, noisy, or unannotated [12, 13, 14].

In the context of LAMMPS, synthetic data generation can involve curating DSL-conformant script elements to create randomized but executable '.in' files. These files reflect the structure and interdependencies of real simulations, while enabling large-scale experimentation with input variation. For example, combinations of atom styles, boundary types, and pairwise interaction parameters can be synthesized to construct diverse training sets, including both common and edge-case configurations.

This approach not only enhances the training corpus for LLMs but also supports robust evaluation. Since synthetic data is fully controllable, researchers can define target outputs and error conditions to systematically probe a model's ability to generalize from prompt instructions to correct LAMMPS syntax and logic.

Recent advances have shown that large models fine-tuned on synthetic code data exhibit notable improvements in syntax fidelity and downstream performance. Furthermore, synthetic data lends itself well to curriculum learning paradigms [15], wherein models are gradually exposed to tasks of increasing complexity—from basic lattice generation to full NPT ensemble simulations. This scaffolding approach helps LLMs form stable internal representations of the LAMMPS domain.

Beyond fine-tuning, synthetic data also facilitates prompt design and RAG module evaluation. By generating thousands of valid-natural language pairs, researchers can better test how retrieval-based augmentations affect completion quality. The synthetic prompt corpus can include parameterized questions, command explanations, and multi-step simulation workflows, offering a rich environment for instruction tuning.

In this thesis, a hybrid methodology is used to construct synthetic LAMMPS scripts: brute-force assembly of simulation elements is combined with rule-based logic to ensure domain coherence. Scripts are validated using LAMMPS runtime checks and manual inspection, forming a curated dataset for model training and benchmarking.

2.4 Prompt Engineering

Prompt engineering has become a foundational technique in improving the performance of large language models (LLMs) across a wide range of code generation tasks. Rather than modifying model parameters, prompt engineering involves carefully crafting the input text—known as the prompt—to guide the model's output toward a desired form. This approach is especially critical for domain-specific languages like LAMMPS, where syntactic correctness and structural logic are vital.

Simple prompting techniques form the basis of many initial experiments in code generation. Research has shown that including structured metadata such as copyright notices, library import statements, or file headers in the prompt can improve the syntactic accuracy and completeness of generated code [3]. Conversely, misleading or ambiguous annotations like "TODO" comments have sometimes been found to reduce model performance, leading to incomplete or erroneous outputs [16]. Contextual cues such as "Imagine you are a software engineer..." have also been shown to activate more creative or structured outputs, enhancing models' awareness of broader context [17].

Prompt performance is also sensitive to prompt length, word order, and phrasing. Seemingly minor changes—such as adjusting question length or reordering instructions—can yield measurable differences in the accuracy and fluency of generated code [18]. These findings reinforce the insight that LLMs are fundamentally next-token predictors; the quality of their output is deeply influenced by the linguistic signals embedded in the input.

Beyond basic template tuning, reasoning-based prompting techniques have emerged as a powerful paradigm for improving LLM reasoning and code synthesis. Chain-of-Thought (CoT) prompting asks the model to articulate its intermediate reasoning steps before producing a final output. This approach has been widely adopted in logic and arithmetic tasks but has also proven valuable in guiding LLMs through multistep code generation processes [19]. Building on CoT, the Chain-of-Code (CoC) method encourages models to produce pseudocode-like planning sequences that mirror human strategies for decomposing programming tasks.

Structured prompting methods like Structured Chain-of-Thoughts (SCoT) further enhance LLMs' capacity to reflect program logic. Here, the model is guided to organize its generation according to program structure, explicitly commenting on which blocks perform specific functions or how modules relate to one another. This technique enhances the interpretability and maintainability of generated scripts.

More advanced prompting frameworks use symbolic representations like trees or graphs to model divergent reasoning paths. Tree-of-Thoughts (ToT) enables models to explore multiple possible code solutions in parallel, evaluating and selecting from them based on intermediate scoring heuristics. Graph-of-Thoughts (GoT) generalizes this process by treating thought units as graph nodes, allowing the LLM to flexibly compose and combine reasoning paths into a coherent solution.

Verification-enhanced prompting offers another layer of reliability. Chain-of-Verification (CoVe) methods prompt the model to critically assess its own outputs, such as verifying abstract syntax tree (AST) structure or checking for logical consistency. This process has been shown to significantly increase the rate of syntactically valid and executable code generations [20].

Finally, prompt optimization can itself be approached as a search or learning problem. Techniques such as automatic prompt selection, gradient-guided optimization, or prompt ensembling can be used to identify the most effective prompts for a given task or model. These multi-stage pipelines often treat prompt engineering as an iterative design process, with feedback loops that test and refine prompt variants for performance gains [21, 22].

In the context of this thesis, prompt engineering serves as one of the primary strategies for improving LLM generation of LAMMPS input files. By leveraging both intuitive design and data-driven optimization, this approach seeks to align model behavior with the domain-specific requirements of molecular dynamics simulation scripting.

2.5 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is a hybrid framework that integrates information retrieval with language model generation. Originally proposed by Lewis et al., RAG was developed to improve the performance of large language models on knowledge-intensive tasks by equipping them with access to an external memory in the form of a document corpus [23]. A retriever model identifies the most relevant textual fragments from this corpus based on a given query, and a generator model then incorporates this information to produce coherent and informed outputs.

In the RAG framework, the retriever—commonly implemented using dense passage retrieval (DPR)—is responsible for fetching the top-k relevant documents from the external knowledge base. The generator, often a model like BART, uses this retrieved content to guide generation. Lewis et al. explored both static document conditioning—where the same documents are used for an entire response—and

dynamic conditioning—where different parts of the response can incorporate information from different documents.
In this thesis, a similar retrieval paradigm is applied, where the retriever is decoupled from the generator and used to access a curated corpus of LAMMPS documentation, example scripts, and simulation tutorials. This provides the LLM with relevant, up-to-date information, enhancing its ability to produce domain-specific and structurally valid input scripts without requiring full retraining.
RAG has seen increasing adoption in the code generation domain, where hallucinations—incorrect or misleading code outputs—are a persistent challenge. Recent work by Wang et al. demonstrated that RAG frameworks can support automated code repair by retrieving patch fragments from a trusted codebase and feeding them into a CodeT5 generator to produce revised, functional code [24]. Parvez et al. introduced REDCODER, a system that leverages repositories like GitHub and Stack Overflow to retrieve semantically relevant code examples, which are then refined by LLMs to suit the current prompt [25].
Other frameworks like ReACC and ProCC have pushed RAG techniques further by introducing customized prompt templates and adaptive retrieval strategies. ProCC, in particular, incorporates three types of prompts—semantically enriched, hypothetical, and summarization-based—to improve code completion tasks. Its adaptive retrieval selection algorithm, built on a contextual bandit strategy using LinUCB, allows dynamic selection of the most appropriate retrieval strategy based on prompt content and retrieval history [26].
Although most RAG applications in code generation have focused on general-purpose programming languages, emerging evidence suggests that DSLs may benefit even more from such approaches. DSLs are often low-resource and underrepresented in pretraining corpora, making them fertile ground for retrieval-based enhancements. Baumann et al. investigated the use of RAG in a novel DSL generated via MontiCore, showing that LLMs could synthesize correct code for unseen DSL constructs when retrieval was properly integrated [27].
Zhou et al. extended this idea through DocPrompting, a retrieval framework that identifies documentation snippets aligned with a given natural language intent. These snippets are appended to the prompt and help steer LLM generation within the bounds of syntax and semantic rules for a specific programming language [28]. In the context of LAMMPS, DocPrompting-style RAG could enable generation grounded in user manuals, parameter references, and real usage patterns—offering more structured support than conventional fine-tuning.
Overall, RAG offers a powerful complement to prompt engineering and fine-tuning. It enables scalable, memory-augmented generation while reducing the risks of hallucination and misinformation. For low-resource domains like LAMMPS scripting, RAG acts as a lightweight yet effective method to incorporate authoritative, domain-specific knowledge into the generation process without extensive retraining.

2.6 Fine-Tuning

In scenarios where only limited training data is available, fine-tuning large language models (LLMs) remains a viable and effective strategy for enhancing their performance on specialized tasks such as code generation. Rather than relying solely on prompt engineering or large-scale retraining, fine-tuning on small, carefully curated datasets can significantly improve a model's ability to generalize in domain-specific or low-resource environments. This is particularly relevant for generating input scripts in domain-specific languages (DSLs) like those used in LAMMPS.
Several recent studies have explored the benefits of fine-tuning with small datasets. Chen et al. [6] and Nijkamp et al. [29] found that even limited amounts of domain-aligned training data can sub-

stantially improve the syntactic correctness and semantic consistency of generated code. Instruction tuning methods such as Self-Instruct [30] and parameter-efficient approaches like LoRA [31] have further reduced the resource burden associated with fine-tuning, making this approach accessible for academic or applied research settings.

When it comes to DSLs or scientific code, synthetic data generation serves as a powerful complement to fine-tuning. Prior work by Kulal et al. [12] and Austin et al. [13] has demonstrated that combining real-world examples with synthetically generated inputs can help bootstrap learning in code synthesis tasks. These studies confirm that small, diverse, and task-relevant training sets can guide LLMs to internalize structural rules and logic patterns even for DSLs that were underrepresented during pretraining.

In this thesis, fine-tuning is applied using a hybrid corpus composed of manually curated, rule-based, and synthetically generated LAMMPS input files. While the dataset size is relatively small, its quality and diversity are designed to expose the LLM to a wide range of simulation structures and parameter combinations. Fine-tuning on this dataset aims to enhance the model's understanding of LAMMPS syntax, reduce hallucinated commands, and improve the executability of generated scripts.

3 Methods

3.1 Overview of Experimental Pipeline

This research aims to assess and enhance the performance of large language models (LLMs) in generating syntactically correct and semantically valid LAMMPS input scripts from natural language descriptions. The study adopts a modular experimental framework composed of a baseline evaluation and three enhancement strategies: prompt engineering, retrieval-augmented generation (RAG), and fine-tuning. Each strategy is applied to the same set of natural language prompts to systematically evaluate improvements.
The pipeline is initiated by a user-defined prompt describing a molecular dynamics (MD) simulation scenario (e.g., "simulate a Lennard-Jones fluid at 300 K using NVT ensemble"). The input prompt then proceeds through one of the following generation strategies:
Baseline Generation: A zero-shot configuration using a pretrained LLM (e.g., GPT-3.5 or GPT-4) without additional guidance. **Prompt Engineering:** Augments the prompt using structural templates, few-shot examples, chain-of-thought (CoT) reasoning, or code scaffolding techniques, enabling the model to better capture the logic of LAMMPS scripting [19, 3]. **Retrieval-Augmented Generation (RAG):** Injects contextually relevant documentation or code snippets retrieved from an indexed knowledge base constructed from LAMMPS manuals, GitHub projects, and example scripts [23, 28].
Fine-Tuning: Utilizes a compact yet diverse dataset of labeled prompt-script pairs to update model weights, aligning its generation capability with LAMMPS syntax and semantics [30, 31].
All generated input scripts are evaluated based on three key criteria:

- **Syntactic Correctness:** Whether the script adheres to LAMMPS DSL syntax and passes a structural parse.
- **Executability:** Whether the script runs successfully on the LAMMPS engine without fatal errors.
- **Simulation Validity:** Whether the setup is physically meaningful and consistent with the prompt's intent.

These evaluation metrics enable a quantitative and qualitative comparison between generation strategies.

3.2 Benchmark Dataset Construction

A benchmark dataset was curated to support both model training (fine-tuning) and evaluation. The dataset consists of natural language prompts paired with valid LAMMPS input scripts, derived from three complementary sources to ensure coverage, accuracy, and diversity:
LAMMPS Documentation. The official LAMMPS manual contains a variety of example inputs showcasing core features and syntax usage. Snippets and full scripts were programmatically extracted and verified for correctness.
GitHub Repositories. Input scripts from public LAMMPS-based projects hosted on GitHub were mined and filtered. These reflect authentic research practices and domain-specific modeling techniques not always covered by documentation.
Synthetic Rule-Based Scripts. A rule-based generator was developed to construct new LAMMPS scripts by combining fundamental commands (e.g., `lattice`, `region`, `create_atoms`, `pair_style`)

figures/experiment_pipeline.pdf

Figure 1: High-level overview of the experimental pipeline: natural language prompt $\rightarrow$ optimization strategy (Baseline, Prompt Engineering, RAG, Fine-Tuning) $\rightarrow$ LAMMPS input script $\rightarrow$ evaluation.

into varied and structurally valid configurations. This allowed augmentation of the dataset with edge cases and underrepresented command combinations [12, 13].

Natural language prompts were either manually composed or generated using structured templates that reflect common simulation tasks. Each prompt describes the physical system, desired simulation method (e.g., NVE/NVT), and any relevant constraints or material models. The resulting dataset serves as a foundation for prompt testing, retrieval indexing, and fine-tuning.

3.3 Baseline Model and Evaluation Protocol

The baseline model used in this study is OpenAI's GPT-3.5 Turbo, selected for its widespread adoption and strong performance in zero-shot code generation. As an optional upper-bound reference, GPT-4 is also tested on a limited subset of tasks to assess the performance ceiling of current general-purpose LLMs. All baseline experiments are conducted under zero-shot settings, where the model receives only the raw natural language prompt without any additional contextual examples or domain-

specific instructions.
To evaluate the quality of generated LAMMPS scripts, we define three criteria:

- **Syntactic Correctness:** The script must conform to LAMMPS input syntax and pass initial parsing.
- **Executability:** The script must run successfully without crashing or returning syntax-related errors when executed in LAMMPS.
- **Simulation Validity:** The output of the simulation must match physical expectations given the prompt, such as stable energy behavior, valid boundary conditions, or appropriate use of force fields.

Each criterion is assessed through a mix of automated tools (e.g., syntax validators, runtime logs) and manual inspection. The results from the baseline serve as the reference for subsequent evaluation of optimization techniques.

3.4 Prompt Engineering Setup

Prompt engineering is applied to guide the LLM's generation more effectively, leveraging both structural patterns and task reasoning. Several prompting strategies are tested:
Few-Shot Prompting: Prompts include multiple example pairs of natural language descriptions and their corresponding LAMMPS input segments. This strategy provides the model with concrete grounding and format expectations.
Chain-of-Thought (CoT): Prompts are designed to encourage the model to reason through the generation process by explicitly stating intermediate planning steps. For example, the prompt may instruct the model to first decide on atom style, then define regions, and finally configure fixes [19].
Chain-of-Code (CoC): A variant of CoT where the model generates pseudocode or scaffolded steps before completing the full script. This encourages decomposition of the simulation setup process into interpretable parts.
Structured Templates: Prompts use a fixed structure (e.g., problem description $\rightarrow$ constraints $\rightarrow$ expected output format) to reduce variability in responses and increase reproducibility.
Ablation studies are conducted by comparing script accuracy and executability across different prompt formats. The results highlight the sensitivity of LLMs to prompt structure and support findings from prior work that LLM behavior is significantly influenced by input phrasing and logic structure [3, 18].

3.5 Retrieval-Augmented Generation (RAG) Setup

The Retrieval-Augmented Generation (RAG) module in this research enhances model performance by incorporating relevant LAMMPS documentation and examples into the input context. The retrieval system is implemented using LangChain integrated with a FAISS-based vector index for fast similarity search. Embeddings for documents are generated using a Sentence-BERT model fine-tuned for code and technical language.
The retrieval corpus consists of:

- **LAMMPS Official Manual:** Including descriptions, usage notes, and command syntax examples.
- **Example Scripts:** Extracted from GitHub repositories and research papers to reflect realistic use cases. reference

- **Q&A-like Pairs:** Generated manually to simulate Stack Overflow-style entries for troubleshooting and configuration explanation.

Given a user prompt, the top-k most relevant documents (typically $k = 3$) are retrieved using cosine similarity over the embedding space and concatenated to the original prompt. This enriched prompt is then passed to the language model for generation. This setup allows the model to access domain-specific knowledge without requiring architectural changes or full retraining [23, 28].

3.6 Fine-Tuning Configuration

For the fine-tuning strategy, we use a pretrained GPT-style model (based on EleutherAI's GPT-NeoX 6B) as the base. Fine-tuning is performed on a hybrid dataset consisting of manually curated, rule-based, and synthetically generated LAMMPS input files. Each sample in the dataset is paired with a natural language prompt describing the intended simulation setup.
Preprocessing includes:

- Tokenization using the GPT-compatible tokenizer.
- Prompt-script pair formatting in instruction-tuning style.
- Filtering malformed scripts using a LAMMPS linter and runtime checks.

Training parameters:

- Epochs: 3–5
- Batch size: 4
- Optimizer: AdamW
- Learning rate: 2×10^{-5}
- Loss: Cross-entropy on token-level prediction

For parameter-efficient fine-tuning, we also experiment with LoRA (Low-Rank Adaptation) [31]. LoRA reduces training overhead by injecting low-rank trainable matrices into selected attention layers, significantly lowering memory usage. The fine-tuning is performed using Hugging Face's PEFT (Parameter-Efficient Fine-Tuning) library.

3.7 Combined Methods

To explore the complementarity of techniques, we test combinations of methods:

- **Prompt Engineering + RAG:** The prompt structure (e.g., CoT) is enhanced with retrieved documentation, allowing the model to reason step-by-step with explicit support.
- **Fine-Tuning + CoT Prompting:** The fine-tuned model is evaluated under CoT-style prompts to assess whether domain-specific knowledge further enhances reasoning quality.
- **Fine-Tuning + RAG:** We also examine whether a fine-tuned model benefits from retrieval, or whether the fine-tuning process makes retrieval redundant.

These combined experiments aim to reveal interaction effects between optimization strategies and identify which combinations yield the highest performance gains in terms of syntax accuracy, code executability, and simulation alignment.

4 Experimental Setup

4.1 Experiments

The experiments are designed to evaluate the performance of large language models (LLMs) across different code generation strategies—baseline, prompt engineering, retrieval-augmented generation (RAG), fine-tuning, and their combinations—on a diverse set of LAMMPS simulation prompts. Each strategy is tested using the same set of benchmark prompts to ensure comparability.
The goal of the experiments is to answer the following questions:

- How effective are LLMs at generating executable and valid LAMMPS input files in a zero-shot setting?
- To what extent does prompt engineering improve code correctness and executability?
- Can retrieval from domain-specific documentation enhance model outputs?
- Does fine-tuning with a small but diverse dataset improve generation quality?
- Which combination of strategies yields the best performance?
- some commonly seen error from generated input scripts

The experimental results are analyzed both quantitatively and qualitatively. Quantitative results are reported based on the evaluation metrics described below, while qualitative analysis focuses on error types and user satisfaction with prompt-response alignment.

4.2 Data

The benchmark dataset used for the experiments is composed of 300 natural language prompts and their corresponding LAMMPS input scripts. These prompts were either derived from actual simulation projects or generated using structured templates, representing a range of simulation scenarios such as NVE/NVT ensembles, solid-state models, and thermostatted systems.
The paired scripts originate from three main sources:

- Official LAMMPS documentation and example directories.
- Publicly available GitHub repositories with valid LAMMPS workflows.reference
- Rule-based and synthetic script generators built for this project.

The dataset is divided into two parts:

- **Evaluation Set:** 200 prompt-script pairs used to measure model performance across all methods.
- **Fine-Tuning Set:** 100 examples reserved for supervised fine-tuning and instruction tuning.

Scripts are verified using LAMMPS execution, and prompts are curated to reflect real-world use cases and domain-relevant phrasing.

4.3 Metrics

To measure the effectiveness of each generation strategy, the following evaluation metrics are employed:

- **Syntactic Correctness:** Evaluates whether the generated script adheres to valid LAMMPS input syntax. This is assessed using a custom syntax validator and manual reviews.
- **Executability:** Measures whether the generated script runs without runtime errors in a sandboxed LAMMPS environment. This binary metric (pass/fail) reflects basic usability.
- **Simulation Validity:** Assesses whether the script produces physically meaningful results. This includes checks for stable energy behavior, convergence, or matching the prompt's expected ensemble or material.
- **Edit Distance:** Computes the token-level edit distance between the generated script and the reference (ground truth) script. It approximates similarity in structure and expression.
- **BLEU Score:** The Bilingual Evaluation Understudy (BLEU) score, adapted for code, quantifies the n-gram overlap between generated and reference scripts. A higher BLEU score indicates greater textual similarity and helps evaluate generation fluency.

4.4 Hyperparameters

For fine-tuning experiments, we implemented two common strategies: full-parameter fine-tuning and parameter-efficient adaptation using LoRA (Low-Rank Adaptation).
Full fine-tuning: Model: GPT-NeoX 6B
Batch size: 4
Learning rate: 2×10^{-5}
Optimizer: AdamW
Epochs: 3 to 5 depending on convergence
Loss: Token-level cross-entropy

LoRA fine-tuning:
LoRA rank: 8
LoRA alpha: 16
Dropout: 0.1
Trainable layers: attention projection matrices only

5 Results

Table 1: Performance comparison of GPT-4.0 Turbo with different enhancement strategies

Model Strategy	Runnable (%)	F1 Total	F1 Average	BLEU Score
GPT-4.0 (Base)	12.523	0.622	0.585	41.219
GPT-4.0 + CoT	15.004	0.718	0.695	47.824
GPT-4.0 + RAG	17.234	0.732	0.702	49.545
GPT-4.0 + CoT + RAG	19.834	0.792	0.763	54.345

Table 2: Impact of Prompting and RAG across Dataset Types-Full Fine-Tuning

CoT	RAG	Fine-tuning Dataset	Runnable (%)	F1 Total	F1 Average	BLEU Score

Table 3: Impact of Prompting and RAG across Dataset Types-LoRA Fine-Tuning

CoT	RAG	Fine-tuning Dataset	Runnable (%)	F1 Total	F1 Average	BLEU Score

Table 4: Fine-Tuning Strategy Comparison (Full vs. LoRA)

Fine-Tuning Strategy	Runnable (%)	F1 Total	F1 Average	BLEU Score
Full Fine-Tuning				
LoRA Fine-Tuning				

6 Conclusion

6.1 Summary of Main Contributions

6.2 Future Work

Bibliography

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[3] A. D. White and J. McDaniel, "Generative ai for quantum chemistry: Input generation and configurational space exploration with large language models," *Digital Discovery*, vol. 2, no. 4, pp. 946–956, 2023.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[8] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[9] D. Hendrycks, S. Burns, H. Kadavath, A. Arora, S. Basart, D. Tang, D. Song, J. Steinhardt, and J. Zou, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[10] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[11] C. Xu, D. Zhang, S. Choudhury, B. Y. Lin, Y. Wu, A. Mehta, Y. Huang, R. Tang, S. Sun, K.-W. Chang, *et al.*, "A survey of large language models for scientific discovery," *arXiv preprint arXiv:2307.05420*, 2023.

[12] Y. Kulal, K. Yang, S. Zhang, and P. Liang, "Spoc: Search-based pseudocode to code," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[13] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Chen, M. Litwin, P. Barham, *et al.*, "Program synthesis with large language models," in *International Conference on Learning Representations (ICLR)*, 2021.

[14] K. Lakhotia, A. Jain, R. Jain, D. Tarlow, R. Puri, S. R. Choudhury, A. Svyatkovskiy, and N. Sundaresan, “Data engineering for improving deep learning in code generation,” in *NeurIPS*, 2021.

[15] Y. Wang, C. Zhang, W. Chen, B. Zhou, and M. Huang, “Self-prompting large language models for code generation,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022.

[16] S. O’Brien, S. Lee, S. Choudhary, and M. Pradel, “The impact of misleading comments on code generation models,” in *IEEE Symposium on Security and Privacy (SP)*, 2022.

[17] A. Koul, I. Nejadgholi, and D. Magazzeni, “Prompting gpt-3 to be reliable.” arXiv preprint arXiv:2205.12543, 2022.

[18] Y. Wang, D. Khashabi, R. Y. Pang, and Y. Choi, “On the impact of prompt length, word order, and verbosity on language models,” in *Proceedings of NAACL*, 2022.

[19] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” *arXiv preprint arXiv:2201.11903*, 2022.

[20] S. Yao, Y. Zhao, D. Yu, S. S. Yu, D. Zhou, Y. Fu, *et al.*, “Tree of thoughts: Deliberate problem solving with large language models,” *arXiv preprint arXiv:2305.10601*, 2023.

[21] P. J. Liu, M. Saleh, E. A. Pot, *et al.*, “Lost in the middle: How language models use long contexts,” *arXiv preprint arXiv:2307.03172*, 2023.

[22] T. Shin, Y. Razeghi, R. Logan, E. Wallace, and S. Singh, “Autoprompt: Eliciting knowledge from language models with automatically generated prompts,” in *Proceedings of EMNLP*, 2022.

[23] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, I. Kulikov, V. Chaudhary, A. Fan, S. Bhosale, *et al.*, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” in *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[24] C. Wang, S. Liu, D. Zhang, G. Neubig, F. Niu, and B. Y. Lin, “Learning code repair with retrieval-augmented edit transformers,” in *Findings of ACL*, 2022.

[25] M. R. Parvez, J. Guo, N. I. Hassan, D. Wang, M. A. S. Islam, X. Hu, and et al., “Retrieval augmented code generation and summarization,” in *arXiv preprint arXiv:2108.04092*, 2021.

[26] K. Zheng, D. Wang, J. Liu, and K.-W. Chang, “Procc: Progressive code completion with retrieval-augmented prompt learning,” in *ICLR*, 2023.

[27] A. Baumann, S. Shah, A. Schürr, *et al.*, “Retrieval-augmented generation for unseen dsls: An empirical study,” in *International Conference on Automated Software Engineering*, 2023.

[28] Q. Zhou, Z. Liu, H. Jiang, X. Du, Y. Li, B. Y. Lin, and G. Neubig, “Docprompting: Generating code by retrieving relevant documentation,” *arXiv preprint arXiv:2210.16514*, 2022.

[29] E. Nijkamp, D. Ha, Y. Tu, Z. Lin, B. Wang, Y. Xu, S. Cao, R. Jain, Y. Shao, Z. Zhu, *et al.*, “Codegen: An open large language model for code with multi-turn program synthesis,” *arXiv preprint arXiv:2203.13474*, 2022.

[30] Y. Wang, D. Khashabi, C. Zhang, X. Li, S. Cahyawijaya, W. Chen, X. Li, P. Lu, K. Thaker, Y. Xue, *et al.*, "Self-instruct: Aligning language models with self-generated instructions," *arXiv preprint arXiv:2212.10560*, 2023.

[31] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations (ICLR)*, 2022.

Appendices

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

A More Stuff To Say

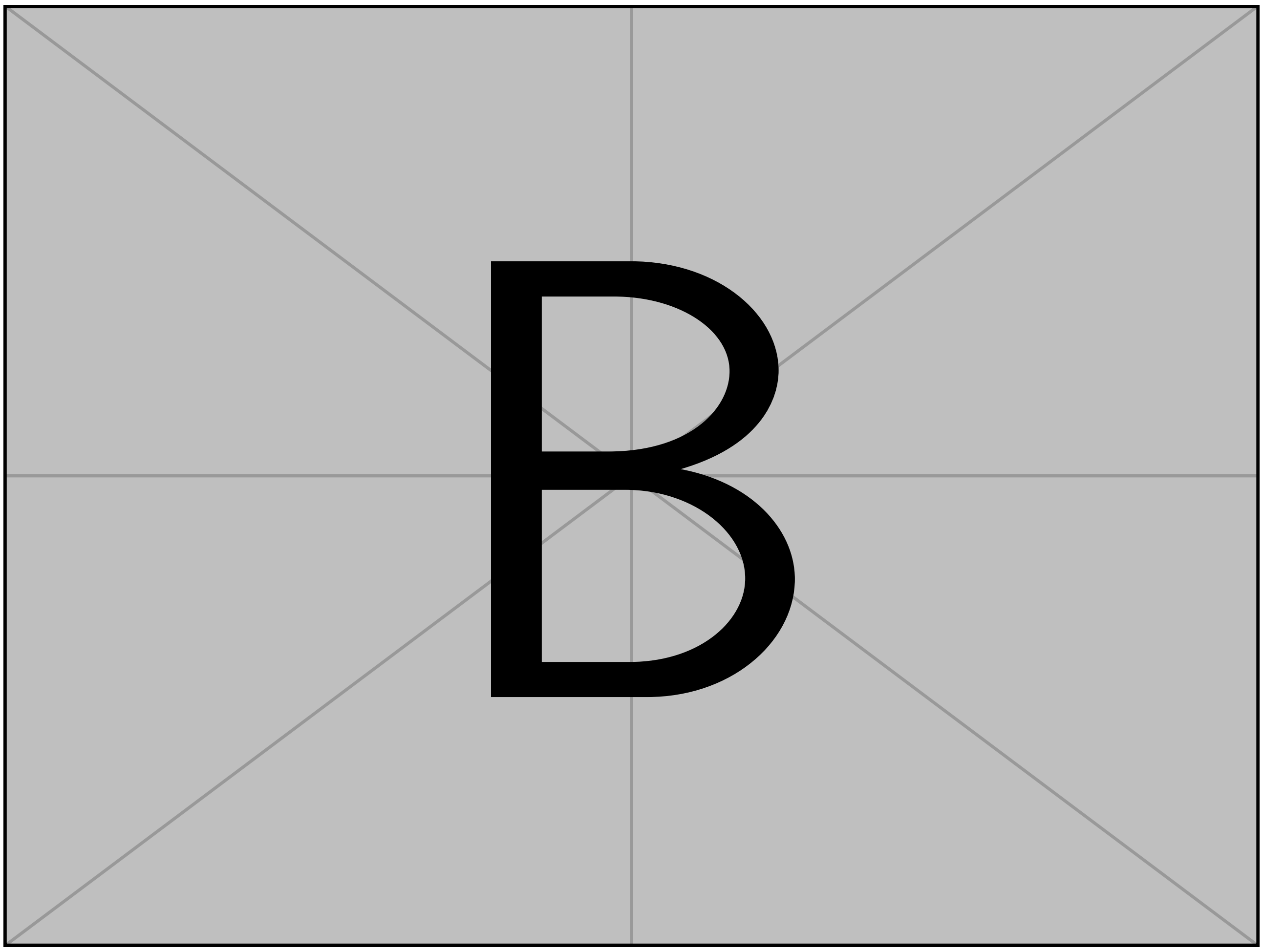

Figure 2: Example of an image in the appendix

B Even More Stuff To Say

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

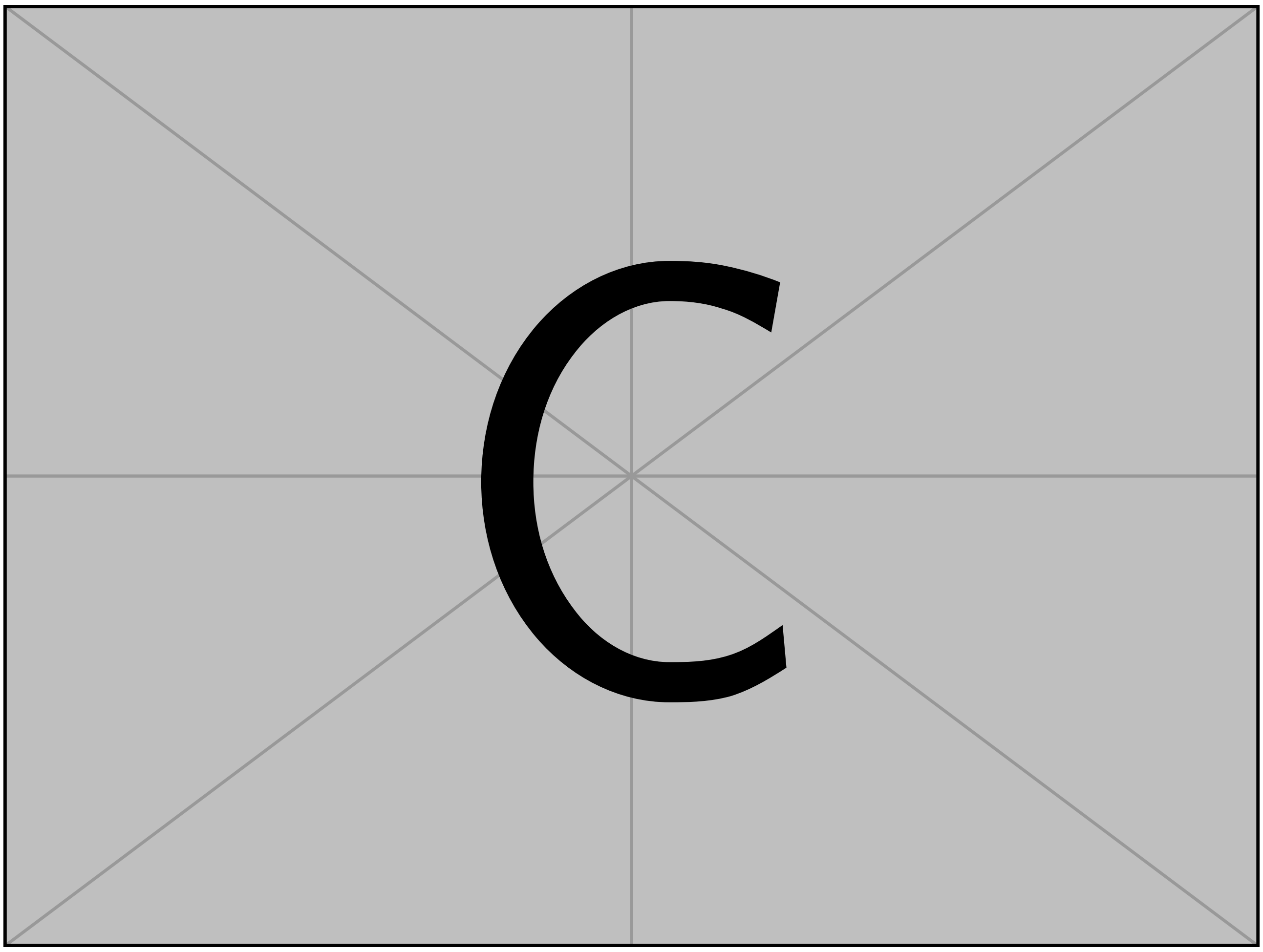

Figure 3: Example of an image in the appendix