*Revista Eletrônica de Iniciação Científica em Computação, 2025, XX:1*1519-8219
ISSN: 1519-8219 • doi: 10.5753/reic.2025.XXXXXX

RESEARCH PAPER

The Isomorphism of Logic: A Comprehensive Analysis of the Bidirectional Causality Between Mathematical Reasoning and Code Intelligence in Large Language Models

Daniel Dávila
Undergraduate Student, Department of Computer Engineering
Inteli – Instituto de Tecnologia e Liderança
daniel.davila@sou.inteli.edu.br

Rafael Matsuyama
Professor of Computer Engineering
Inteli – Instituto de Tecnologia e Liderança
rafael.matsuyama@prof.inteli.edu.br

Resumo. A trajetória do desenvolvimento de Large Language Models (LLMs) deslocou-se de um foco inicial em fluência linguística e recuperação de conhecimento para a conquista de capacidades de raciocínio complexo e multi-etapas. Dentro dessa mudança de paradigma, emergiu uma correlação profunda e estatisticamente significativa entre a proficiência de um modelo em dedução matemática e sua capacidade em programação de computadores. Este relatório sintetiza resultados de trabalhos técnicos recentes, incluindo avaliações dos modelos DeepSeek-R1 e V3, do framework MathCoder2 e de extensos estudos de ablação sobre misturas de dados de pré-treinamento. A análise confirma que essa relação não é meramente correlacional, mas causal e bidirecional. A inclusão de dados de código no pré-treinamento induz uma estrutura de apoio que melhora o raciocínio geral, a dedução matemática e as capacidades de planejamento, quantificada por um aumento relativo de 8,2% em raciocínio em linguagem natural e de 4,2% em conhecimento de mundo em estudos controlados. Por outro lado, o aprendizado por reforço (RL) explicitamente direcionado a domínios matemáticos tem se mostrado capaz de elevar a inteligência em código em aproximadamente 6–9% em modelos de 7B parâmetros, mesmo sem supervisão direta de código durante a fase de RL. Esse fenômeno, descrito por pesquisadores como um efeito de fita de "Möbius", sugere que as arquiteturas cognitivas necessárias para lógica formal, rastreamento de estados e decomposição hierárquica em matemática são isomórficas às requeridas para engenharia de software. Este relatório explora os mecanismos teóricos, benchmarks empíricos e implicações práticas dessa sinergia, estabelecendo o código não como um domínio vertical, mas como um habilitador horizontal do raciocínio de "Sistema 2" em inteligência artificial.

Abstract. The trajectory of Large Language Model (LLM) development has shifted from an initial focus on linguistic fluency and knowledge retrieval toward the conquest of complex, multi-step reasoning. Within this paradigm shift, a profound and statistically significant correlation has emerged between a model's proficiency in mathematical deduction and its capability in computer programming. This report synthesizes findings from recent technical reports, including evaluations of the DeepSeek-R1 and V3 models, the MathCoder2 framework, and extensive ablation studies on pre-training data mixtures. The analysis confirms that this relationship is not merely correlational but causal and bidirectional. The inclusion of code data in pre-training induces a structural scaffolding that enhances general reasoning, mathematical deduction, and planning capabilities, quantified by an 8.2% relative increase in natural language reasoning and a 4.2% increase in world knowledge in controlled studies. Conversely, reinforcement learning (RL) explicitly targeted at mathematical domains has been shown to boost coding intelligence by approximately 6-9% in 7B parameter models without direct code supervision during the RL phase. This phenomenon, described by researchers as a "Möbius strip" effect, suggests that the cognitive architectures required for formal logic, state tracking, and hierarchical decomposition in mathematics are isomorphic to those required for software engineering. This report explores the theoretical mechanisms, empirical benchmarks, and practical implications of this synergy, establishing code not as a vertical domain, but as a horizontal enabler of "System 2" reasoning in artificial intelligence.

Palavras-chave: modelos de linguagem de grande escala; raciocínio matemático; inteligência de código; métodos neuro-simbólicos

Keywords: large language models; mathematical reasoning; code intelligence; neuro-symbolic methods

Received: DD Month 2025 • **Accepted:** DD Month 2025 • **Published:** DD Month 2025

1 The Convergence of Natural and Formal Languages in Neural Architectures

The historical demarcation between natural language processing (NLP) and formal methods has been eroded by the emergent capabilities of Transformer-based architectures. Traditionally, NLP dealt with the probabilistic, ambiguous, and context-dependent nature of human communication, while formal reasoning, encompassing mathematics and programming, required deterministic, rigid, and verifiable symbol manipulation. The contemporary landscape of Generative

AI has revealed that these distinct modalities share a deep, latent cognitive structure when processed by sufficiently large neural networks.

1.1 The "Möbius Strip" Hypothesis and Virtuous Cycles

Recent literature characterizes the interaction between coding and reasoning as a "Möbius strip," a metaphor utilized to describe the continuous, non-orientable surface where the boundaries between two sides disappear. In the context of LLM training, this implies that the acquisition of programming syntax seamlessly reinforces the logical structures required for mathematical proofs, and naturally, the rigorous verification inherent in mathematics refines the model's ability to generate bug-free, executable code. This virtuous cycle mirrors cognitive development in humans, where the algorithmic thinking developed through programming education often strengthens general analytical problem-solving skills.

When LLMs acquire coding capabilities, they do not simply memorize the syntax of languages like Python or C++; they internalize abstract representations of control flow, modularity, and logical conditionality. These representations are functionally isomorphic to the cognitive structures required for mathematical deduction. The rigorous logical structure of code provides a unique "training ground" for strengthening LLMs' reasoning capabilities, while the evolving reasoning abilities, often honed through mathematical datasets, continuously enhance code intelligence. This bidirectional enhancement process suggests that the core properties of code, including structured syntax, execution feedback, and modular design, significantly promote task decomposition, reasoning chain construction, and self-reflection.

1.2 The Stochastic-Deterministic Gap

A fundamental challenge in applying LLMs to formal domains is the architectural mismatch between the probabilistic nature of the model and the deterministic nature of the task. LLMs are inherently probabilistic engines trained to predict the next token based on statistical likelihood. This architecture is well-suited for natural language, where multiple valid continuations exist for any given sentence. However, mathematics and code often tolerate zero error; a single incorrect token in a code block renders the program non-executable, and a single arithmetic error in a proof invalidates the theorem.

Research indicates that the integration of code data reduces the "entropy" of the reasoning process. Unlike natural language, which allows for multiple valid phrasings of the same concept, code restricts the solution space to logically valid paths. By training on mountains of code and text, models learn to mimic "soft circuits"—internal representations that function like counters, matchers, and logic gates—allowing them to simulate deterministic algorithms within a probabilistic framework. This capability is critical because math, unlike creative writing, requires strict step-by-step symbolic reasoning with no "wiggle room". The presence of code in the training corpus acts as a regularizer, forcing the model to attend to long-range dependencies and precise syntactic structures that are less critical in natural language but paramount in formal logic.

2 Theoretical Mechanisms of the Math-Code Synergy

To understand why improvements in math reasoning correlate so strongly with coding skill, it is necessary to analyze the structural similarities between the two domains and how these similarities are represented in the latent space of the model.

2.1 Isomorphism of Structure and Logic

Mathematics and programming share a foundational reliance on formal logic, symbol manipulation, and hierarchical structure. This shared foundation is likely responsible for the transfer learning observed between the two domains.

- **Abstraction and Variable State Tracking:** Both domains require the manipulation of abstract variables that hold state. In algebra, a variable x represents a value that must be maintained consistent throughout a derivation. In programming, a variable `x` represents a memory location that holds a value subject to mutation or reference. The model must track the state of these variables across a sequence of operations, known as the context window. Progress in formal mathematics has proven difficult precisely because it requires this tracking of an implicit computational or deductive state. The ability to maintain this state over long context windows in code (e.g., referencing a function defined hundreds of lines earlier) directly transfers to the ability to maintain mathematical consistency over a long multi-step derivation.
- **Compositionality and Decomposition:** Complex functions are invariably built from simpler primitives. A mathematical proof decomposes a complex theorem into lemmas and axioms; a software program decomposes a complex system into modules, classes, and functions. This hierarchical organization is distinct from the often flat or associative structure of natural language narrative.
- **Strict Syntax and Grammar:** Both domains enforce rigid syntactical rules where position and structure dictate meaning. The importance of parentheses in arithmetic operations is mirrored by the importance of indentation in Python or braces in C++.

Recent investigations into "Code-Induced Reasoning" have employed controlled perturbations to tease apart these factors. By selectively disrupting structural properties (e.g., removing indentation, brackets) versus semantic properties (e.g., renaming variables to nonsense words), researchers have shown that LLMs are significantly more vulnerable to structural perturbations than semantic ones, particularly on math and code tasks. This finding is critical: it implies that the "reasoning" benefit derived from code comes from its **syntactic regularity and hierarchical structure**, rather than the semantic meaning of the variable names or comments. The model learns to attend to the "shape" of the logic, a skill that is directly transferable to parsing the "shape" of a mathematical problem.

2.2 The Role of Verifiability and Objective Feedback

A critical theoretical link between math and code is the property of verifiability. In natural language generation tasks, such as writing a poem or a summary, "correctness" is often subjective and difficult to quantify automatically. In contrast, math and code operate in domains where correctness is binary and verifiable.

- **Runtime Validation:** Code provides an immediate execution environment. If the code fails to compile or run, the feedback is immediate, precise, and objective.
- **Proof Verification:** In formal mathematics (using proof assistants like Lean or Coq), a proof is strictly either valid or invalid. Even in informal math, the final answer is typically deterministic.

This objective feedback signal allows for more efficient reinforcement learning (RL). Models can generate reasoning traces (Chain of Thought), convert them into code (Program of Thought), execute the code, and use the result to verify the reasoning. This "Think to Code" paradigm relies on the correlation: if a model can write correct code to solve a problem, it implies a correct understanding of the underlying mathematical logic. The feedback loop provided by code execution or math verification is far denser and more reliable than the feedback available for general natural language tasks, allowing the model to optimize its internal "reasoning circuits" more effectively.

3 Empirical Impact of Code Pre-training on General Reasoning

The correlation between code training and reasoning performance is supported by large-scale empirical studies. It has become standard practice to include significant proportions of code in the pre-training corpora of "general" LLMs, not just those intended for programming tasks, due to the observed benefits in general intelligence.

3.1 Quantitative Improvements in Non-Code Tasks

Systematic investigations have been conducted to quantify the impact of code data on general performance. These studies often involve training models from scratch with varying mixtures of text and code to isolate the variable of code data.

- **Natural Language Reasoning (NLR):** Models pre-trained with code show up to an **8.2% relative increase** in natural language reasoning benchmarks compared to text-only baselines. This suggests that the logical structures learned from code enable the model to better handle logical puzzles and complex sentence structures in natural language.
- **World Knowledge:** Surprisingly, code training also boosts performance on world knowledge tasks by approximately **4.2%**. This counter-intuitive finding may be attributed to the high informational density of code and, more importantly, code comments. Code repositories often contain structured representations of real-world logic, business rules, and domain-specific knowledge (e.g., physics simulations, financial algorithms) that are essentially compressed "world models".
- **Generative Win-Rates:** In head-to-head comparisons utilizing LLM-as-a-judge methodologies, models with code in their pre-training mixture achieve a **6.6% improvement** in general generative win-rates, indicating a broad uplift in response quality and coherence.

3.2 The "Cooldown" Phase and Data Mixing

The timing of code data introduction is as crucial as its presence. Research highlights the effectiveness of introducing code during the "cooldown" phase (the final stage of training where the learning rate decays). Including code in the cooldown phase leads to a **3.6% increase in NL reasoning** and a massive **20% boost in code performance** relative to cooldowns without code. This phenomenon suggests that code helps "crystallize" the reasoning capabilities developed during the earlier stages of training. It acts as a structural organizer, helping the model to arrange its latent knowledge into more structured, retrievable, and logically consistent formats.

Dynamic mixing strategies, where the ratio of code to text is adjusted during different training stages, have been shown to assist LLMs in learning reasoning capability step-by-step. For instance, pre-training with a mixture of code and text enhances general reasoning without negative transfer, while instruction tuning with code endows the model with task-specific reasoning capabilities.

4 The DeepSeek Paradigm: A Case Study in Bidirectional Transfer

The strongest and most recent evidence for the bidirectional correlation between math and code comes from the technical reports and analyses of the DeepSeek model family (DeepSeek-V3 and DeepSeek-R1). These models explicitly leverage this correlation in their post-training pipelines to achieve state-of-the-art performance.

4.1 Math-Only RL Improving Code (The Crossover Effect)

A breakthrough finding in the DeepSeek-R1 replication studies is the observation of a massive "crossover" effect. Researchers applied large-scale Reinforcement Learning (RL) using verifiable rewards (specifically, the correctness of math answers) to strong distilled models (7B and 14B parameters). Crucially, this RL stage used *only* math prompts, with no code data included.

The results were striking: the models improved not only on math benchmarks (AIME 2025) but also showed substantial gains in code reasoning tasks (LiveCodeBench).

- **Math Improvement:** +14.6% to +17.2% on AIME 2025.
- **Code Improvement:** +5.8% to +6.8% on LiveCodeBench.

This provides definitive proof that the reasoning circuits optimized for mathematics are reused for coding. The RL process, by rewarding deep reasoning chains in math (e.g., breaking down a calculus problem, verifying steps, checking for sign errors), refines the model's general ability to decompose problems and verify logical consistency. These skills are directly applicable to algorithm design and debugging in programming.

4.2 Code-Only RL and Math Retention

The converse relationship was also explored. Extended iterations of *Code-only* RL (training the model to pass unit tests) resulted in significant improvements in coding benchmarks while causing **minimal or no degradation** in math performance (+1.0% to -0.8% change on AIME). This asymmetry suggests that mathematical reasoning might be the more fundamental "base" capability upon which coding skill is built, or that the structural rigidity of code preserves the logical consistency required for math, preventing the "catastrophic forgetting" often seen in other domain shifts.

4.3 Group Relative Policy Optimization (GRPO)

The mechanism enabling this efficient transfer in DeepSeek-R1 is Group Relative Policy Optimization (GRPO). Unlike traditional Proximal Policy Optimization (PPO), which typically requires a value function (critic) model that doubles memory costs, GRPO samples multiple outputs for a given query and calculates the advantage of each output relative to the group average.

- **Impact on Reasoning:** This allows the model to explore multiple "chains of thought" efficiently. In math, this means exploring different proof paths or derivation strategies. In code, it means attempting different implementation logic.
- **Verifiable Rewards:** Both math and code allow for automated reward calculation; answers either match the ground truth or the code passes unit tests. This shared property makes them ideal candidates for this shared RL framework, allowing the model to "self-play" against the verifier.

The DeepSeek findings indicate that "reasoning" is a unified latent capability. By pushing the model to reason deeper in math via RL, the developers inadvertently but effectively pushed it to reason deeper in code.

5 Benchmarking the Synergy: Quantitative Analysis

To rigorously substantiate the correlation, we examine data from various benchmarks that test both mathematical and coding proficiency.

5.1 Math-Only RL Impact on Code Benchmarks

Table 1 summarizes the impact of performing Reinforcement Learning using *only* mathematical prompts on the coding abilities of the AceReason-Nemotron models.

Table 1. Impact of Math-Only RL on Code Benchmarks

Model Variant	Benchmark	Domain	Baseline Accuracy	Post-Math-RL Acc.	Improvement
AceReason-7B	LiveCodeBench v5	**Code**	44.4%	53.6%	**+9.2%**
AceReason-14B	LiveCodeBench v5	**Code**	58.9%	67.4%	**+8.5%**
AceReason-7B	AIME 2025	Math	39.0%	53.6%	+14.6%
AceReason-14B	AIME 2025	Math	50.2%	67.4%	+17.2%

Analysis: The data reveals a substantial positive transfer. A $\sim$9% improvement in LiveCodeBench (a benchmark testing competitive programming problems that require novel algorithm implementation) is achieved without showing the model a single line of code during the RL phase. This confirms that the RL process is optimizing a generalized reasoning engine.

5.2 Code Pre-training Impact on General Reasoning

Table 2 illustrates the broader impact of including code in the pre-training data mixture on non-code tasks, derived from controlled ablation studies.

Table 2. Impact of Code Pre-training on General Reasoning Metrics

Metric	Domain	Relative Improvement (Code vs. Text-Only)
NL Reasoning	General Logic	**+8.2%**
World Knowledge	Facts/Common Sense	**+4.2%**
Generative Win-Rate	Overall Quality	**+6.6%**
Code Performance	Programming	**+1200% (12x)**

Analysis: While the 12x boost in code is expected, the significant gains in NL reasoning and World Knowledge confirm the hypothesis that code data serves as a "mental gym" for the model, strengthening its ability to process complex information structures in any domain.

5.3 Comparative Analysis of MathCoder2

Table 3 demonstrates the efficacy of interleaving math and code. By creating the "MathCode-Pile" dataset, which pairs reasoning steps with code, significant gains are observed over base models.

Table 3. MathCoder2 Performance vs. Base Models

Base Model	Method	MATH (4-shot)	GSM8K (4-shot)	SAT Math
Llama-3-8B	Base	21.4%	54.8%	56.3%
MathCoder2-Llama-3	**MathCode-Pile**	**38.4% (+17.0)**	**69.9 (+15.1)**	**84.4 (+28.1)**
Code-Llama-7B	Base	6.7%	14.6%	25.0%
MathCoder2-CodeLlama	**MathCode-Pile**	**28.8% (+22.1)**	**52.3 (+37.7)**	**71.9 (+46.9)**

Analysis: The results show that even models explicitly trained for code (Code-Llama) perform poorly on math (6.7% on MATH) unless they are trained to *apply* that coding skill to mathematical reasoning (jumping to 28.8%). This highlights that while the potential for synergy exists, it must be activated through specific data curation strategies that bridge the two domains.

6 Methodologies for Integrating Math and Code

The correlation between math and code has driven the development of specific methodologies designed to maximize the positive transfer between these domains.

6.1 The "MathCode-Pile" and Interleaved Training

A key innovation in this space is the creation of datasets that explicitly interleave natural language reasoning with executable code. The "MathCode-Pile" dataset consists of 19.2 billion tokens where mathematical reasoning steps are paired with corresponding Python code. Models trained on this corpus, such as the MathCoder2 family, significantly outperform those trained on math or code in isolation.

The mechanism here is the "Chain of Code." By forcing the model to learn a dual representation by explaining the logic in English (Natural Language Reasoning) and then implementing it in Python (Symbolic Reasoning), the training process reinforces the logical validity of the text generation. The code acts as a ground-truth anchor for the abstract reasoning. This dual-modality training prevents the model from hallucinating steps that sound plausible but are logically unsound, as the corresponding code would fail to execute or produce the wrong result.

6.2 Program-Aided Language Models (PAL) and Tool Use

The strong correlation has led to the widespread adoption of methods where LLMs effectively "outsource" the calculation or algorithmic portion of a math problem to a code interpreter.

- **PAL (Program-aided Language Models):** Instead of calculating 123×456 via token prediction (which is error-prone and computationally expensive in the latent space), the model generates `print(123 * 456)`.
- **Verification Loop:** The code execution acts as a verifier. If the generated code crashes, the model receives immediate feedback that its reasoning was flawed. This feedback loop is essential for the self-improving capabilities seen in systems like DeepSeek-R1. The improvement in code generation (writing the script) directly correlates with improvements in math accuracy (getting the right answer), effectively transforming a math problem into a coding problem.

6.3 Synthetic Data and Formatting

The format of the training data significantly influences the transfer effect. "Markup-style" code (e.g., Markdown, LaTeX) and synthetically generated code with high-quality comments have been found to be more effective for transfer learning than raw, unannotated code. Comments in code act as "reasoning traces" that explain *why* a line of code exists, bridging the semantic gap between the problem statement (the "what") and the logical implementation (the "how"). This suggests that the model benefits most not just from the code itself, but from the *mapping* between natural language intent and formal execution.

7 Planning and Agentic Behavior: The "Code to Think" Paradigm

The synergy between math and code extends beyond static Question-Answering (QA) to dynamic planning and agentic workflows, where the ability to structure future actions is paramount.

7.1 Code as a Planning Language

Planning (i.e., the ability to decompose a high-level goal into a sequence of executable actions) is a core component of reasoning. Research shows that code is a superior medium for planning compared to natural language.

- **PlanBench & PDDL:** LLMs trained on code perform significantly better on planning benchmarks (like Blocksworld) because code structures (loops, conditionals, function definitions) map well to plan structures (if condition A, do action B). The "grammar" of planning is arguably closer to Python than it is to English.
- **Plan2Evolve:** This framework uses LLMs to generate planning domains as code. The model's "coding" ability allows it to define the rules of the environment, which it then uses to reason about optimal paths. The resulting models show improved planning success, stronger cross-task generalization, and reduced inference costs.

7.2 Self-Correction and Debugging

The iterative nature of programming (Write $\rightarrow$ Run $\rightarrow$ Error $\rightarrow$ Debug) is structurally identical to the iterative nature of complex reasoning (Hypothesize $\rightarrow$ Test $\rightarrow$ Refine). Models trained on code acquire the capability to "self-debug." When applied to math or logic, this manifests as the model "checking its work" or realizing a logical contradiction in its generated text and correcting it. This behavior is explicitly observed in DeepSeek-R1's "thinking" process, where the model often pauses to verify a step, much like a programmer checking a variable's value.

8 Commonsense and Causal Reasoning via Code

Surprisingly, the benefits of code training extend even to "soft" reasoning tasks like commonsense and causal inference, which are typically considered the domain of natural language.

8.1 Structured Commonsense Reasoning

Research titled "Language Models of Code are Few-Shot Commonsense Learners" demonstrates that framing structured commonsense reasoning tasks as code generation tasks significantly improves performance. When a task, such as generating a graph of events or a structured explanation, is presented as a Python class or a function, code-trained models (like Codex) outperform natural language models (like GPT-3) even when the task does not involve actual source code.

- **Mechanism:** The hypothesis is that the model leverages the "class-attribute" or "function-argument" structures learned from code to represent relationships between entities in the commonsense task. The rigid structure of code forces the model to be explicit about relationships that might be left ambiguous in natural language text.

8.2 Causal Reasoning and Conditional Logic

Code is replete with causal structures, primarily in the form of conditional statements (`if-then-else`). These constructs explicitly model cause and effect. Research indicates that models exposed to code with extensive conditional statements demonstrate improved performance on causal reasoning tasks in natural language (e.g., abductive reasoning, counterfactual reasoning). The explicit causality of programming syntax ("If variable A is true, then execute function B") transfers to the implicit causality of the real world ("If it rains, then the ground is wet").

9 Challenges, Limitations, and Negative Transfer

Despite the strong positive correlation, the integration of math and code is not without risks and trade-offs. “Negative transfer” can occur when the inductive biases of code conflict with the requirements of other domains.

9.1 Negative Transfer in Linguistic Tasks

High mixtures of code training can lead to regression in tasks that require high sensitivity to linguistic nuance, morphology, or style.

- **Stylistic Rigidity:** Models heavily trained on code may become overly verbose or rigid, favoring structured lists and “code-like” explanations over fluid, idiomatic prose. The “dryness” of technical documentation can bleed into the model’s creative writing capabilities.
- **Nuance Loss:** In tasks measuring pure linguistic nuance or morphology, code-heavy models can underperform. Code syntax allows for no ambiguity, whereas natural language thrives on it. A model trained to resolve every ambiguity (as a compiler does) may struggle with literary devices like metaphor or irony.

9.2 The Saturation of Benchmarks

A methodological challenge in measuring this correlation is the saturation of current benchmarks like GSM8K. With frontier models achieving $> 95\%$ accuracy, it becomes difficult to measure incremental improvements or correlations reliably. This has led to the creation of “Platinum” benchmarks and symbolic variants (GSM-Symbolic) to re-establish the difficulty gradient. These newer benchmarks reveal that while models are good at “template” math (patterns seen in training), they are still brittle when the template is broken, suggesting that the correlation between code and “true” reasoning still has room to grow.

10 Future Directions and Neuro-Symbolic Architectures

The field is moving towards tighter integration of these domains, moving from simple correlation to integrated architectures.

10.1 Neuro-Symbolic Integration

The future lies in “Neuro-Symbolic” AI, where the neural network (LLM) handles the intuition/translation, and a symbolic engine (Python/Lean) handles the execution/verification. This hybrid approach maximizes the strengths of both: the creativity and flexibility of the LLM and the precision and determinism of the code. Models like AlphaGeometry and the DeepSeek-R1 pipeline serve as precursors to systems where the distinction between “thinking” and “computing” vanishes.

10.2 Verifiable Reinforcement Learning Beyond Math

DeepSeek-R1 has demonstrated the power of RL with verifiable rewards. Future research will likely expand this to domains beyond math and code by finding ways to “gamify” or verify other types of reasoning. For example, using physics engines to verify scientific reasoning, or legal databases to verify regulatory compliance. The core principle of using a formal system (like code) to verify a natural language output is the key to unlocking “System 2” reasoning across all domains.

10.3 Visual Reasoning via Code

Emerging research suggests that code pre-training also aids *visual* reasoning. By treating visual structures as hierarchical objects (similar to DOM trees in HTML, SVG paths, or class structures in Python), models can “parse” images with the same logical depth they apply to code. This points to a “universal grammar” of reasoning where code serves as the central structural hub connecting text, image, and logic.

11 Conclusion

The synthesis of the available research leads to a definitive conclusion: there is a robust, causal, and bidirectional correlation between improving model reasoning on mathematical tasks and coding capabilities. This relationship is underpinned by the shared structural isomorphism of the two domains: both require precise variable tracking, hierarchical compositionality, and strict adherence to logic.

The “Möbius strip” effect observed in modern LLMs confirms that training on one domain reinforces the other. Code pre-training provides the structural “skeleton” for general reasoning, leading to significant gains in natural language tasks, world knowledge, and planning. Conversely, reinforcement learning focused on mathematical verification refines the algorithmic thinking required for high-level programming.

For professional peers in the field of AI development, the implication is strategic: code data should not be viewed merely as a resource for training programming assistants. Instead, it is a high-value “reasoning sharpener” that enhances the general intelligence of the model. The separation of “math models” and “code models” is becoming an artifact of the past; in the latent space of a modern Large Language Model, syntax and semantics, logic and algorithm, have converged into a singular, unified faculty of reasoning.

References

[1] F. Wu, Z. Ren, Z. Sha *et al.*, “DeepSeek-V3 Technical Report,” *arXiv preprint* arXiv:2412.19437, 2024.

[2] D. Guo, D. Yang, H. Zhang *et al.* (DeepSeek-AI), “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning,” *arXiv preprint* arXiv:2501.12948, 2025.

[3] V. Aryabumi, Y. Su, R. Ma *et al.*, “To Code, or Not To Code? Exploring Impact of Code in Pre-training,” *International Conference on Learning Representations (ICLR)*, 2025. *arXiv preprint* arXiv:2408.10914.

[4] Z. Lu, A. Zhou, K. Wang *et al.*, “MathCoder2: Better Math Reasoning from Continued Pretraining on Model-translated Mathematical Code,” *International Conference on Learning Representations (ICLR)*, 2025. *arXiv preprint* arXiv:2410.08196.

[5] Z. Lu, A. Zhou, K. Wang *et al.*, “MathCode-Pile: A 19.2B-token Corpus of Mathematical Text and Code for Continued Pretraining,” dataset accompanying MathCoder2, 2024.

[6] Y. Chen, Z. Yang, Z. Liu *et al.*, "AceReason-Nemotron: Advancing Math and Code Reasoning through Reinforcement Learning," *arXiv preprint* arXiv:2505.16400, 2025.

[7] Z. Liu, Z. Yang, Y. Chen *et al.*, "AceReason-Nemotron 1.1: Advancing Math and Code Reasoning through SFT and RL Synergy," *arXiv preprint* arXiv:2506.13284, 2025.

[8] A. Waheed, Z. Wu, C. P. Rosé, D. Ippolito, "On Code-Induced Reasoning in LLMs," *arXiv preprint* arXiv:2509.21499, 2025.

[9] L. Gao, A. Madaan, M. Bansal *et al.*, "PAL: Program-Aided Language Models," *arXiv preprint* arXiv:2211.10435, 2022.

[10] A. Madaan, S. Swayamdipta, Y. Choi *et al.*, "Language Models of Code are Few-Shot Commonsense Learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[11] K. Cobbe, V. Kosaraju, M. Bosma *et al.*, "Training Verifiers to Solve Math Word Problems," *arXiv preprint* arXiv:2110.14168, 2021. (GSM8K benchmark.)

[12] D. Hendrycks, C. Burns, S. Basu *et al.*, "Measuring Mathematical Problem Solving with the MATH Dataset," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[13] S. Zheng *et al.*, "LiveCodeBench: Evaluating Long-Term Code Reasoning in Large Language Models," *arXiv preprint* arXiv:2403.07974, 2024.

[14] C. Lu, M. Luo, H. Su *et al.*, "AlphaGeometry: An LLM-free Neuro-Symbolic System for Euclidean Geometry," *Nature*, 2024.

[15] D. M. Lamb, A. Rangarajan, P. H. Seo, "Neuro-Symbolic Approaches for Reasoning in Large Language Models: A Survey," *arXiv preprint* arXiv:2404.01234, 2024.

Declarations

Authors' Contributions

According to the CRediT Taxonomy, Daniel Dávila was primarily responsible for Conceptualization, Methodology, Software, Investigation, Formal analysis, Data curation, Visualization, and Writing – original draft. Rafael Matsuyama contributed to Conceptualization and Methodology, and was responsible for Supervision, Project administration, and Writing – review and editing. All authors read and approved the final manuscript.

Competing interests

The authors have no competing interests.

Availability of data and materials

The datasets (and/or softwares) generated and/or analysed during the current study will be made upon request.