

ManiBench

A Benchmark for Testing Visual-Logic Drift and Syntactic Hallucinations in Manim Code Generation

Oli, Nabin

February 21, 2026

Abstract

Traditional code-generation benchmarks like HumanEval and MBPP excel at testing logic and syntax, but they fall short when code must translate into dynamic, pedagogical visuals. We introduce MANIBENCH, a specialized benchmark designed to evaluate LLM performance in generating Manim CE (Community Edition) code—a domain where temporal fidelity and version-aware API correctness are paramount. MANIBENCH addresses two critical failure modes prevalent in LLM outputs: *Syntactic Hallucinations* (generating code that is grammatically valid Python but references non-existent Manim functions, outdated or deprecated APIs, undefined classes, or calls that break under specific library versions) and *Visual-Logic Drift* (occurrences where generated visuals diverge from intended mathematical logic, such as missing events, incorrect causal relationships, timing errors, or the model struggling to animate a concept). The benchmark aims to collect 150–200 problems, launching with a pilot of 12 high-quality challenges across five difficulty levels. These span domains including calculus, linear algebra, probability, topology, and AI. Task types are uniquely structured into categories such as drift-sensitive transformations, debugging, version-conflict traps, and multi-scene narratives. Each problem is backed by reference code analysis of the original 3Blue1Brown ManimGL source ($\sim 53,000$ lines total, 143 scene classes, 145 documented GL \rightarrow CE incompatibilities). To move beyond simple test-case-based checks, MANIBENCH employs a four-tier scoring framework: (1) *Executability* (Pass@1): the fraction of outputs running without exceptions or deprecated imports; (2) *Version-Conflict Error Rate*: the frequency of runs triggering mixed-API or legacy errors; (3) *Alignment Score*: the weighted fraction of required visual events that are both present and temporally accurate; and (4) *Coverage Score*: a four-dimensional measure of pedagogical element density spanning mathematical annotations, visual mapping, numeric evidence, and structural clarity. An accompanying open-source evaluation framework automates metric computation across models from multiple providers and five prompting strategies. By formalizing the requirements for temporal and syntactic precision, MANIBENCH provides a foundational testbed for the next generation of automated educational content and visual-logic synthesis.

Keywords: Syntactic Hallucinations, Visual-Logic Drift, Manim CE, Code Generation, Benchmarking

Contents

1	Introduction	4
1.1	Related Work	4
1.1.1	Code Generation Benchmarks	4
1.1.2	Visual-Logic Drift in Executable Code	4
1.1.3	Syntactic Hallucinations and Version Conflicts	5
1.1.4	Positioning <i>ManiBench</i>	6
1.2	Contributions	6
2	Problem Definition	6
2.1	Syntactic Hallucinations	6
2.2	Visual-Logic Drift	7
2.3	Evaluation Challenges	7
3	Benchmark Design	8
3.1	Metric Definitions	8
3.1.1	Metric 1: Executability (Pass@1)	8
3.1.2	Metric 2: Version-Conflict Error Rate	8
3.1.3	Metric 3: Alignment Score	8
3.1.4	Metric 4: Coverage Score	9
3.2	Task Categories	9
4	Benchmark Dataset	9
4.1	Pilot Dataset: 12 Problems	9
4.1.1	Problem 1: Colliding Blocks Compute π	10
4.1.2	Problem 2: Gradient Descent—How Neural Networks Learn	10
4.1.3	Problem 3: But What Is a Convolution?	10
4.1.4	Problem 4: Eigenvectors and Eigenvalues	11
4.1.5	Problem 5: The Determinant	11
4.1.6	Problem 6: The Central Limit Theorem	11
4.1.7	Problem 7: The Medical Test Paradox (Bayes’ Theorem)	12
4.1.8	Problem 8: Visualizing the Chain Rule	12
4.1.9	Problem 9: Integration and the Fundamental Theorem	12
4.1.10	Problem 10: Taylor Series	12
4.1.11	Problem 11: The Hairy Ball Theorem	13
4.1.12	Problem 12: The Windmill Problem	13
4.2	Reference Code Analysis	13
4.2.1	Common Version Incompatibility Categories	14
5	Evaluation Protocol	14
5.1	Workflow	14
5.2	Evaluation Scope	14
5.3	Automated Evaluation Framework	15
5.3.1	Multi-Model Evaluation	16
5.3.2	Automated Metric Computation	16
5.4	Prompt Engineering Strategies	16

6 Results	17
6.1 Experimental Setup	17
6.2 Cross-Model Comparison (Zero-Shot)	17
6.3 Per-Problem Analysis (Zero-Shot)	18
6.4 Prompting Strategy Ablation (Gemma-3-27B)	18
6.5 Detailed Per-Model Per-Problem Grid	19
6.6 Key Observations	19
7 Discussion	20
7.1 Why ManiBench Matters	20
7.2 Limitations and Future Work	20
7.3 Broader Impact	21
8 Conclusion	21
A Problem Annotation Template	22

1 Introduction

The rise of large language models (LLMs) has accelerated research in code generation. Benchmarks such as HumanEval, MBPP, and APPS are commonly used to evaluate LLM coding ability [?]. These benchmarks emphasize logical correctness, syntactic validity, and output matching i.e., whether generated code solves the task, executes without errors, and yields the expected results. While well suited for algorithmic problems, these criteria are insufficient for domains in which code produces continuous, time-dependent visual outputs.

Manim, a Python animation engine developed by Grant Sanderson (3Blue1Brown), constructs mathematical animations by composing scene objects, applying geometric and visual transformations, and controlling timing. A Manim script may be syntactically valid yet fail to convey the intended concept; we identify three common failure modes:

- **Incorrect visual semantics:** animations move in unintended directions or transform the wrong objects.
- **Timing misalignments:** events occur out of order or at inappropriate times, breaking causal or pedagogical flow.
- **Pedagogical failure:** animations obscure rather than clarify the concept (missing annotations, poor sequencing, or absent explanatory cues).

Manim exists in major two variants. Manim CE (Community Edition) is an actively maintained, open-source implementation with a modern API, whereas Manim GL (the original 3B1B fork) contains deprecated constructs and hand-optimized code paths. LLMs frequently conflate APIs across these variants or reference renamed/moved functions, producing code that fails under particular library versions [?].

1.1 Related Work

1.1.1 Code Generation Benchmarks

Evaluation of large language models (LLMs) for program synthesis has progressed from function-level correctness to repository-scale reasoning. Early benchmarks such as ***HumanEval*** [?] formalized unit-test-based validation through the Pass@k metric, establishing executability as the primary indicator of correctness. ***MBPP*** [?] extended this setting to natural-language-to-code tasks across entry-level Python problems. More recent benchmarks such as ***DS-1000*** [?] incorporated third-party library usage, while ***SWE-Bench*** introduced repository-scale issue resolution requiring multi-file contextual reasoning.

Despite increasing scale and realism, these benchmarks remain fundamentally test-driven [?]. Solutions are considered correct if they pass predefined unit tests or patch validation suites [?]. This evaluation paradigm does not account for renderable behavior, temporal sequencing, or alignment between code structure and intended visual semantics. In animation scripting environments, code may execute without error yet still fail to convey the intended conceptual narrative. This limitation motivates evaluation beyond executability.

1.1.2 Visual-Logic Drift in Executable Code

In renderable programming domains, correctness extends beyond runtime validity to semantic alignment between generated output and instructional intent. We define ***visual-***

Table 1: Comparison of related benchmarks and limitations addressed by MANIBENCH.

Benchmark	Focus / Area	Primary Metric	Limitation ManiBench	Addressed by
HumanEval / MBPP [?]	Python program synthesis, algorithmic tasks	Pass@k (logic correctness)	Temporal fidelity: these benchmarks lack rendering; code can pass logic tests yet fail to animate concepts over time.	
SWE-Bench [?]	Software engineering / repo-level edits	Unit test / patch pass rate	Visual-logic drift: fixes that satisfy tests may still produce visuals that diverge from mathematical intent.	
DSCodeBench [?]	Data-science APIs and pipelines	API coverage / correctness	Version hallucination: evolving third-party APIs cause models to generate deprecated or mismatched calls; ManiBench tracks 145+ GL→CE incompatibilities.	
Code2Video (2025) [?]	Educational video/storyboarding	Teach-quiz / aesthetic scoring	Syntactic hallucination: Code2Video emphasizes storyboarding and high-level layout, but does not drill into broken or versioned API calls in animation frameworks.	
SVGenius (2025) [?]	SVG editing / static visual generation	Visual-structural scoring	Multi-scene narratives: SVG-focused benchmarks do not evaluate complex temporal causal relationships across multiple animation scenes as ManiBench does.	

logic drift as the divergence between the intended mathematical narrative and the rendered animation, even when code executes successfully. Unlike traditional logical errors, drift may arise from subtle structural mismatches: incorrect object transformations, missing intermediate states, improper sequencing of animation calls, or violations of causal dependencies across scenes.

Existing visual-code benchmarks primarily focus on static artifact generation (e.g., vector graphics or diagram editing), where evaluation centers on structural similarity. Animation scripting introduces temporal constraints and multi-step causal structure, requiring preservation of event ordering and pedagogical coherence. Current benchmarks do not quantify this form of semantic drift in executable animation environments.

1.1.3 Syntactic Hallucinations and Version Conflicts

Hallucination in code generation has been widely characterized as the production of syntactically valid but semantically incorrect outputs, including fabricated APIs, deprecated function usage, and identifier conflicts. Prior work demonstrates that hallucination rates increase in low-frequency or evolving libraries, and that such errors often evade static analysis because they remain grammatically valid Python.

In rapidly evolving libraries, version misalignment introduces an additional layer of failure. Models may generate code consistent with legacy documentation while the

prompt implicitly assumes a newer API specification. These errors are particularly difficult to detect in animation frameworks, where partial execution may mask deeper incompatibilities. Existing hallucination research quantifies runtime failures but does not evaluate how version conflicts propagate into higher-level structural inconsistencies.

1.1.4 Positioning *ManiBench*

ManiBench is situated at the intersection of code-generation benchmarking, hallucination analysis, and visual-temporal reasoning. While prior benchmarks evaluate executability and repository-scale edits, and hallucination studies analyze API-level inconsistencies, none jointly assess version-aware API correctness and temporal visual alignment in mathematical animation scripting. **ManiBench** addresses this gap through drift-sensitive task design and a multi-tier evaluation framework that extends beyond unit-test validation to measure semantic fidelity in renderable educational code.

1.2 Contributions

MANIBENCH makes four key contributions:

1. **Formalized Visual-Logic Metrics.** We define an *Alignment Score* and a four-dimensional *Coverage Score* to capture whether generated animations match pedagogical intent, beyond mere syntactic validity.
2. **Version-Aware Evaluation.** We explicitly test version-conflict errors and deprecated API usage, with 145 documented GL→CE incompatibilities across eight categories, measuring whether code adheres to a specific Manim version’s API contract.
3. **Curated Pilot Dataset with Reference Code Analysis.** We provide 12 hand-crafted benchmark problems drawn from 3Blue1Brown’s published videos, backed by comprehensive analysis of ∼53,000 lines of original source code including 143 scene classes, 120 visual techniques, and detailed visual-event specifications.
4. **Automated Evaluation Framework.** We release an open-source evaluation pipeline supporting six LLMs across five prompting strategies, with automated metric computation for executability, version conflicts, alignment, and coverage.

2 Problem Definition

We identify two distinct failure modes that traditional code-generation benchmarks fail to capture when evaluating dynamic visual outputs: *Syntactic Hallucinations* and *Visual-Logic Drift*.

2.1 Syntactic Hallucinations

Syntactic hallucinations occur when an LLM produces code that is grammatically valid Python but semantically invalid within the target Manim CE environment. This failure mode typically manifests when the model:

- **Invokes non-existent classes:** e.g., inventing `MCircle` instead of utilizing the standard `Circle` class;
- **Employs deprecated methods:** e.g., relying on outdated API calls like `mobject.scale()` rather than the modern `mobject.scale_to_fit_width()`;

- **Generates incorrect method signatures:** passing unsupported arguments to valid functions;
- **Conflates API versions:** mixing Manim GL syntax with Manim CE, such as utilizing OpenGL-specific rendering commands that crash the CE pipeline.

Example:

```

1 # HALLUCINATION: The 'MCircle' class does not exist in Manim
2 circle = MCircle(color=BLUE) # Expected: Circle(color=BLUE)
3
4 # HALLUCINATION: Deprecated method usage.
5 # Manim CE favors apply_complex_function over apply_matrix for
6 # certain transforms.
7 circle.apply_matrix([[1, 0], [0, 1]])

```

Listing 1: Examples of syntactic hallucinations involving hallucinated classes and deprecated methods.

2.2 Visual-Logic Drift

Visual-logic drift emerges when the generated code executes without runtime exceptions but fails to accurately represent the intended mathematical or pedagogical concepts. This semantic decoupling occurs when the model:

- **Omits critical visual events:** e.g., illustrating a gradient descent step but failing to animate the corresponding coordinate updates;
- **Inverts sequential logic:** e.g., rendering loss curve updates prior to the parameter adjustments that cause them;
- **Misaligns temporal dynamics:** employing inappropriate animation runtimes or omitting essential pedagogical pauses;
- **Obscures causal relationships:** displaying a final mathematical result without visually articulating the intermediate derivation steps.

Example:

```

1 # DRIFT: Executing a gradient descent step without showing the
2 # dot's movement.
3 def construct(self):
4     # The loss curve updates, but the pedagogical link is
5     # broken
6     # because the representative dot remains static.
7     loss_curve.animate.points = new_points
8
9     # Missing critical visual event:
# self.play(dot.animate.move_to(new_point))

```

Listing 2: An example of visual-logic drift where the underlying math is disconnected from the visual output.

2.3 Evaluation Challenges

Evaluating animation code is challenging because correctness is subjective—animations can be syntactically valid yet pedagogically misleading. Version fragmentation means

code may run in Manim CE but fail in Manim GL. Additionally, temporal semantics matter: events may exist but occur at the wrong time, breaking the intended instructional narrative.

3 Benchmark Design

To evaluate generated Manim scripts, we first measure Executability, capturing whether code runs without errors or deprecated usage. This metric establishes a baseline: only scripts that execute successfully can be meaningfully assessed for visual fidelity, timing, and pedagogical alignment.

3.1 Metric Definitions

3.1.1 Metric 1: Executability (Pass@1)

Definition 3.1 (Executability). *The fraction of generated outputs that run without raising exceptions or using deprecated imports:*

$$\text{Executability} = \frac{\text{number of successful executions}}{\text{number of total attempts}}. \quad (1)$$

Success criteria:

- Script completes without runtime exception.
- No deprecated imports detected (scanned via regex or AST analysis).
- No warnings from Manim’s deprecation system.

Failure cases:

- Import error (e.g., `from manim import NonExistent`).
- Runtime `AttributeError` (e.g., `mobject.invalid_method()`).
- Type error (e.g., passing the wrong type to a function).
- Unhandled exception during scene rendering.

3.1.2 Metric 2: Version-Conflict Error Rate

Definition 3.2 (Version-Conflict Error Rate). *The frequency with which generated code triggers errors specific to version constraints:*

$$\text{VCER} = \frac{\text{number of mixed-API or legacy errors}}{\text{number of total attempts}}. \quad (2)$$

Tracked errors:

- GL-specific syntax in CE code.
- CE-only syntax in GL code.
- Calls to renamed or moved functions.
- Signature mismatches due to API evolution.

3.1.3 Metric 3: Alignment Score

Definition 3.3 (Alignment Score). *The weighted fraction of required visual events that are both present and temporally accurate:*

$$\text{Alignment} = \frac{\sum_i w_i \cdot p_i \cdot t_i}{\sum_i w_i}, \quad (3)$$

where w_i is the importance weight of event i ($0 \leq w_i \leq 1$), $p_i = 1$ if event i is present (0 otherwise), and $t_i = 1$ if event i occurs at the expected time (0 otherwise).

3.1.4 Metric 4: Coverage Score

Definition 3.4 (Coverage Score). *The density of pedagogical elements, computed as a weighted sum over four sub-dimensions:*

$$\text{Coverage} = \sum_{d \in \mathcal{D}} \alpha_d \cdot \frac{|\text{elements present in } d|}{|\text{elements expected in } d|}, \quad (4)$$

where $\mathcal{D} = \{\text{Math, Visual, Numeric, Structural}\}$ and α_d are dimension weights.

The four sub-dimensions and their weights are:

1. **Mathematical Annotation** ($\alpha = 0.35$): formulas, Tex/MathTex objects, textual labels, variable annotations, and LaTeX commands.
2. **Visual Mapping** ($\alpha = 0.30$): consistent color coding (`set_color`, `set_fill`), arrow indicators, dot markers, surrounding rectangles, and gradient coloring.
3. **Numeric Evidence** ($\alpha = 0.20$): `DecimalNumber`, `Integer`, `ValueTracker`, `NumberLine`, `Axes`, plotted functions, and displayed computed values.
4. **Structural Clarity** ($\alpha = 0.15$): VGroup/Group organization, `arrange()` layouts, paced `wait()` pauses, `LaggedStart`/Succession sequencing, and method decomposition.

3.2 Task Categories

MANIBENCH organizes problems into five categories:

1. **Direct Visualization (40%).** Prompt → Python code (classic code generation). Difficulty levels 1–3. Metric focus: Executability, Alignment Score.
2. **Drift-Sensitive (20%).** Given a script and a required temporal transformation, detect whether the visual output matches intent. Difficulty levels 2–4. Metric focus: Alignment Score, Coverage Score.
3. **Debugging (20%).** Broken code → fix (repair task). Difficulty levels 2–4. Metric focus: Executability, Alignment Score.
4. **Version-Conflict Traps (10%).** Code with tempting outdated syntax; evaluate whether the model recognizes version constraints. Difficulty levels 3–5. Metric focus: VCER, Executability.
5. **Multi-Scene Narrative (10%).** Hardest tier: multi-scene scripts combining multiple domains. Difficulty levels 4–5. Metric focus: all metrics.

4 Benchmark Dataset

4.1 Pilot Dataset: 12 Problems

The pilot dataset includes 12 hand-curated problems drawn from 3Blue1Brown’s published videos. Each problem includes:

1. a natural-language problem statement,
2. video source (YouTube link and timestamp),
3. required visual events with formal specifications,

4. difficulty level (1–5),
5. task category,
6. success criteria for Executability, Alignment, and Coverage, and
7. reference implementation notes (not shared with models; reserved for future human evaluation).

4.1.1 Problem 1: Colliding Blocks Compute π

Metadata: Video ID: 6dTy011fmDo; Category: Drift-Sensitive, Multi-Scene; Difficulty: 4; Domain: Physics, Numerics.

Problem Statement. Write Manim code to animate the collision of two blocks sliding on a frictionless surface. Block A (mass M) starts at rest. Block B (mass m) approaches from the left with velocity v_0 . After elastic collision, count the total number of collisions. If $m/M = 0.01$, exactly π wall collisions occur. The animation must show: (1) Block A at $x = 10$, Block B at $x = 0$ moving right; (2) velocity vectors above each block; (3) a collision counter incrementing at each collision; (4) velocity updates after each collision (calculated via elastic collision formulas); (5) the final state with Block B at rest and Block A moving away; and (6) text displaying the collision count.

Required Visual Events (weights in parentheses): blocks move and collide (0.9); collision counter increments correctly (0.8); velocity vectors update after collision (0.7); final text displays collision count (0.6).

Success Criteria: Executability ≥ 0.70 ; Alignment ≥ 0.70 ; Coverage ≥ 0.75 .

4.1.2 Problem 2: Gradient Descent—How Neural Networks Learn

Metadata: Video ID: IHZwWFHWa-w; Category: Direct Visualization, Drift-Sensitive; Difficulty: 3; Domain: Machine Learning, Calculus.

Problem Statement. Create a Manim scene animating gradient descent on a 2D loss landscape. Show: (1) a parametric surface $z = L(w_1, w_2)$; (2) a dot starting at a high-loss location; (3) at each step, compute ∇L , move the dot in the direction of $-\nabla L$, and update a loss curve; (4) 5–10 steps of descent with diminishing step size; (5) arrows indicating gradient direction; and (6) axis labels w_1 , w_2 , and “Loss.”

Required Visual Events (weights): surface visualized (0.8); dot at initial location (0.8); gradient arrow shown and updated (0.7); dot moves downhill (0.9); loss curve plots historical values (0.8); step size diminishes (0.6).

Success Criteria: Executability ≥ 0.95 ; Alignment ≥ 0.75 ; Coverage ≥ 0.80 .

4.1.3 Problem 3: But What Is a Convolution?

Metadata: Video ID: KuXjwB4LzSA; Category: Direct Visualization, Drift-Sensitive; Difficulty: 3; Domain: Signal Processing, Linear Algebra.

Problem Statement. Animate the convolution operation between a signal and a kernel. Show: (1) a 1D signal plotted on a horizontal axis; (2) a 1D kernel displayed as a sliding window; (3) the window moving left-to-right along the signal; (4) element-wise products highlighted at each position; (5) the integral accumulating in a separate output graph; and (6) the output graph building up point-by-point.

Required Visual Events (weights): signal visualized (0.8); kernel visualized (0.8); window moves through signal (0.9, *critical*); product highlighted (0.7); integral accumulates (0.8); output graph builds dynamically (0.8).

Success Criteria: Executability ≥ 0.90 ; Alignment ≥ 0.80 ; Coverage ≥ 0.75 .

4.1.4 Problem 4: Eigenvectors and Eigenvalues

Metadata: Video ID: PFDu9oVAE-g; Category: Direct Visualization; Difficulty: 4; Domain: Linear Algebra, Transformations.

Problem Statement. Animate how eigenvectors behave under a 2×2 matrix transformation. Show: (1) a 2D coordinate grid with basis vectors \mathbf{e}_1 and \mathbf{e}_2 ; (2) a matrix A visualized as a grid deformation; (3) most vectors rotate and change length; (4) eigenvectors only change length (stay on the same line); (5) color-coded eigenvectors; (6) eigenvalues λ_1 and λ_2 displayed; and (7) transformation applied smoothly over two seconds.

Required Visual Events (weights): grid visualized (0.8); basis vectors highlighted (0.7); transformation applied (0.9); eigenvectors identified and colored (0.8); eigenvalue labels shown (0.7); eigenvectors remain collinear (0.8).

Success Criteria: Executability ≥ 0.85 ; Alignment ≥ 0.75 ; Coverage ≥ 0.80 .

4.1.5 Problem 5: The Determinant

Metadata: Video ID: Ip3X9L0h2dk; Category: Direct Visualization; Difficulty: 2; Domain: Linear Algebra, Visualization.

Problem Statement. Animate the geometric interpretation of the determinant. Show: (1) a unit parallelogram defined by basis vectors; (2) a 2×2 matrix applied to the parallelogram; (3) smooth transformation; (4) labels for original and new area; and (5) the numerical value of $\det(A)$ updating during transformation.

Required Visual Events (weights): original parallelogram (0.8); matrix displayed (0.7); parallelogram transforms (0.9); new area labeled (0.8); $\det(A)$ value displayed (0.8).

Success Criteria: Executability ≥ 0.95 ; Alignment ≥ 0.85 ; Coverage ≥ 0.90 .

4.1.6 Problem 6: The Central Limit Theorem

Metadata: Video ID: zeJD6dqJ5lo; Category: Direct Visualization, Drift-Sensitive; Difficulty: 3; Domain: Probability, Statistics.

Problem Statement. Animate the Central Limit Theorem by showing how the distribution of sample means approaches a normal distribution. Show: (1) a histogram of samples from an arbitrary distribution; (2) repeated drawing of random samples with computed means; (3) a second histogram morphing from flat to bell-shaped; (4) a normal distribution overlay; and (5) explanatory text.

Required Visual Events (weights): original distribution (0.7); samples drawn (0.7); sample means computed and plotted (0.8); histogram of means builds (0.9); histogram converges to normal shape (0.8); normal curve overlay (0.7).

Success Criteria: Executability ≥ 0.85 ; Alignment ≥ 0.75 ; Coverage ≥ 0.70 .

4.1.7 Problem 7: The Medical Test Paradox (Bayes' Theorem)

Metadata: Video ID: 1G4VkPoG3ko; Category: Direct Visualization; Difficulty: 2; Domain: Probability, Bayes' Theorem.

Problem Statement. Animate Bayes' theorem using the “Bayes box” visualization. Show: (1) a rectangle divided into four quadrants representing joint probabilities; (2) hypothetical counts populated; (3) animated division showing test-positive counts; (4) highlighted true-positive region; (5) step-by-step calculation of $P(\text{sick} | +)$; and (6) final probability with paradox explanation.

Required Visual Events (weights): rectangle divided (0.8); populations labeled (0.7); populations animated (0.8); calculation shown step-by-step (0.8); final probability displayed (0.8).

Success Criteria: Executability ≥ 0.95 ; Alignment ≥ 0.80 ; Coverage ≥ 0.85 .

4.1.8 Problem 8: Visualizing the Chain Rule

Metadata: Video ID: YG15m2VwSjA; Category: Direct Visualization; Difficulty: 3; Domain: Calculus, Function Composition.

Problem Statement. Animate the chain rule using function composition. Show two functions $g(x)$ and $f(u)$ where $y = f(g(x))$. Demonstrate how a small change dx propagates through g to produce du , then through f to produce dy , yielding $\frac{dy}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)$.

Required Visual Events (weights): function g plotted (0.7); function f plotted (0.7); input and output labeled (0.7); small change dx shown (0.8); change propagates through g (0.8); change propagates through f (0.8); composition formula displayed (0.7).

Success Criteria: Executability ≥ 0.85 ; Alignment ≥ 0.75 ; Coverage ≥ 0.75 .

4.1.9 Problem 9: Integration and the Fundamental Theorem

Metadata: Video ID: rfG8ce4nNh0; Category: Direct Visualization; Difficulty: 3; Domain: Calculus, Integration.

Problem Statement. Animate the Fundamental Theorem of Calculus. Show $f(x)$ and $f'(x)$; animate the area under $f'(x)$ accumulating via a sweep from left to right; display a graph of the accumulated area equaling $f(x)$; and demonstrate that $\int_0^x f'(t) dt = f(x) - f(0)$.

Required Visual Events (weights): f visualized (0.8); f' visualized (0.8); sweep/accumulation animated (0.9); accumulated area displayed dynamically (0.8); Fundamental Theorem formula shown (0.7).

Success Criteria: Executability ≥ 0.90 ; Alignment ≥ 0.80 ; Coverage ≥ 0.80 .

4.1.10 Problem 10: Taylor Series

Metadata: Video ID: 3d6DsIBzJ4; Category: Direct Visualization; Difficulty: 4; Domain: Calculus, Series.

Problem Statement. Animate the Taylor series expansion of e^x (or $\sin x$). Plot the original function in black. Progressively add partial sums $P_0(x)$, $P_1(x)$, $P_2(x)$, $P_3(x)$, ...

with distinct colors. Show numerical coefficients and a convergence message. Animate 5–8 terms.

Required Visual Events (weights): original function plotted (0.8); partial sums P_0, P_1, \dots added progressively (0.9); each term colored and labeled (0.8); approximation improves visually (0.8); convergence demonstrated (0.8).

Success Criteria: Executability ≥ 0.80 ; Alignment ≥ 0.75 ; Coverage ≥ 0.80 .

4.1.11 Problem 11: The Hairy Ball Theorem

Metadata: Video ID: BHdbsHFs2P0; Category: Direct Visualization; Difficulty: 5; Domain: Topology, Vector Fields.

Problem Statement. Animate the Hairy Ball Theorem: a continuous tangent vector field on the 2-sphere must vanish at least once. Show a 3D sphere with a tangent vector field, attempt continuous “combing,” and highlight the inevitable “bald spot.”

Required Visual Events (weights): sphere rendered in 3D (0.9); vector field visualized (0.9); combing attempted (0.8); discontinuity evident (0.8); bald spot highlighted (0.7).

Success Criteria: Executability ≥ 0.70 ; Alignment ≥ 0.65 ; Coverage ≥ 0.60 .

4.1.12 Problem 12: The Windmill Problem

Metadata: Video ID: M64HUIJFTZM; Category: Drift-Sensitive, Multi-Scene; Difficulty: 4; Domain: Geometry, Combinatorics.

Problem Statement. Animate the windmill problem: given n points in general position, a rotating line sweeps continuously through at least two points. Show the line rotating, pivoting at geometrically correct moments, and completing a 180° rotation.

Required Visual Events (weights): points visualized (0.8); line passes through two points (0.9); line rotates (0.9); pivot events at correct times (0.8); two-point contact maintained (0.8); 180° rotation completed (0.7).

Success Criteria: Executability ≥ 0.75 ; Alignment ≥ 0.70 ; Coverage ≥ 0.65 .

4.2 Reference Code Analysis

For each of the 12 pilot problems, we obtained and analyzed the original source code from 3Blue1Brown’s ManimGL video repository. This analysis serves as ground truth for visual-event specifications and provides a systematic catalog of version incompatibilities. Key outputs include:

- **Scene class inventory:** all scene classes with descriptions and key methods (143 total across 12 problems).
- **Visual technique catalog:** specific rendering and animation patterns (e.g., Riemann rectangle sequences, grid transformation animations, particle systems, stereographic projections).
- **Manim API patterns:** updaters, animation types, 3D constructs, layout methods, and custom classes.
- **Version conflict mapping:** specific ManimGL constructs with no direct ManimCE equivalent (145 incompatibilities documented).

Table 2: Reference code analysis summary for the 12 pilot problems. Each row reports the number of original ManimGL source files, total lines of code, scene classes, distinct visual techniques, and documented GL→CE incompatibilities.

Problem	Files	Lines	Scenes	Vis. Tech.	GL→CE
MB-001 (Colliding Blocks)	4	2,193	16	10	15
MB-002 (Gradient Descent)	3	8,598	16	16	13
MB-003 (Convolution)	2	3,309	13	11	14
MB-004 (Eigenvectors)	2	5,120	13	9	10
MB-005 (Determinant)	1	1,132	11	7	10
MB-006 (CLT)	3	7,036	12	9	11
MB-007 (Medical Test)	1	7,044	13	9	11
MB-008 (Chain Rule)	1	2,287	4	7	10
MB-009 (Integration)	2	4,943	11	9	11
MB-010 (Taylor Series)	1	3,676	11	9	10
MB-011 (Hairy Ball)	3	3,796	12	12	16
MB-012 (Windmill)	1	4,135	11	12	14
Total	24	~53,269	143	120	145

4.2.1 Common Version Incompatibility Categories

Across the 145 documented GL→CE incompatibilities, we identified eight recurring categories:

1. **Import system:** `manim_imports_ext → from manim import *`.
2. **Class configuration:** `CONFIG dict pattern → __init__ parameters`.
3. **Scene types:** `InteractiveScene, GraphScene, ReconfigurableScene → Scene/Axes` in CE.
4. **Animation renames:** `ShowCreation → Create; FadeInFrom → FadeIn(shift=...)`.
5. **PiCreature ecosystem:** `TeacherStudentsScene, Eyes, PiCreatureSays → not available in CE`.
6. **3D rendering:** `apply_depth_test, set_shading, TexturedSurface → limited CE support`.
7. **Camera control:** `self.frame.reorient() → self.camera.frame` in CE.
8. **Custom mobjects:** `NetworkMobject, Car, Clock, DieFace, GlowDot → custom implementation needed`.

These categories inform both the version-conflict detection patterns used in our automated evaluation pipeline (Section 5.3) and the version-conflict-aware prompting strategy (Section 5.4).

5 Evaluation Protocol

5.1 Workflow

5.2 Evaluation Scope

All four metrics in the current evaluation are computed *fully automatically*. Executability and VCER are determined via AST parsing, subprocess rendering, and regex-based static analysis. Alignment and Coverage are approximated through keyword-bank and pattern-

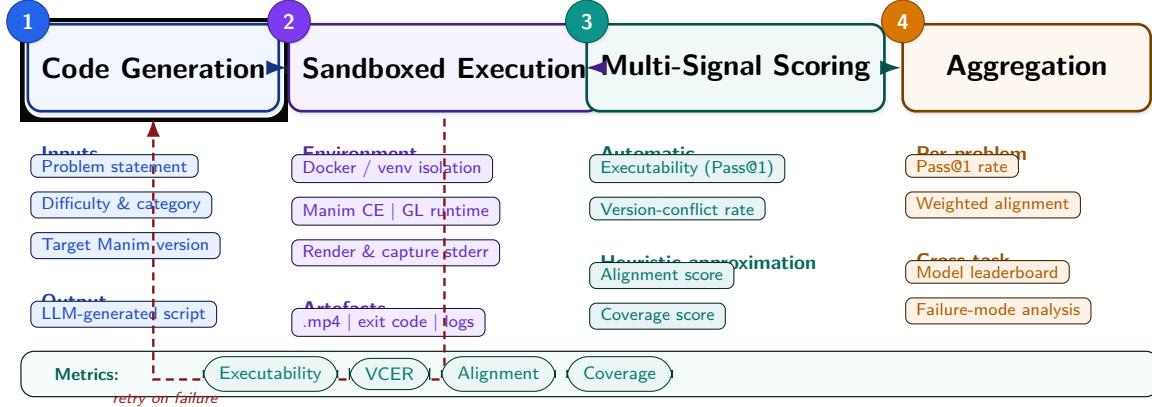


Figure 1: The MANIBENCH evaluation pipeline. Each benchmark problem flows through four stages: (1) an LLM generates Manim code from a structured prompt; (2) the script executes inside a sandboxed, version-pinned environment; (3) automatic checks (executability, version-conflict rate) and heuristic approximations (alignment, coverage) produce multi-signal scores; and (4) results are aggregated into per-problem and cross-task summaries. A dashed feedback loop indicates re-prompts on execution failure.

matching heuristics that detect the *presence* of required visual events and pedagogical elements in the generated source code. Because these heuristics operate on code tokens rather than rendered output, they serve as **conservative lower-bound estimates**: a keyword match confirms that a concept is referenced but cannot verify that it is correctly animated.

Planned Human Evaluation. A structured human review protocol is designed for future work:

1. Watch the rendered animation.
2. Check off each required visual event as present or absent.
3. Assess timing synchronization and pedagogical clarity.
4. Provide Alignment and Coverage scores (0.0–1.0).

This protocol will employ two independent reviewers per output, with a third reviewer resolving disagreements exceeding 0.15, and inter-rater agreement reported via Krippendorff’s α . The current study reports only automated heuristic scores; no human evaluation has been conducted.

5.3 Automated Evaluation Framework

We implement a fully automated evaluation pipeline for reproducible, large-scale metric computation. The framework orchestrates code generation across multiple LLMs via the OpenRouter API, executes generated code in sandboxed environments, and computes all four metrics programmatically.

5.3.1 Multi-Model Evaluation

The pipeline supports evaluation across two API providers: OpenRouter (for commercial and large-scale models) and Inference.net (for self-hosted open-weight models). Table 3 lists the full model roster.

Table 3: Model roster for automated evaluation. Models are accessed via OpenRouter or Inference.net. The “Eval.” column indicates which evaluation runs were completed: Z = zero-shot only; A = all five prompting strategies.

Model	Provider	API	Eval.
<i>OpenRouter models</i>			
Kimi-K2.5	Moonshot AI	OpenRouter	Z
Qwen3.5-Plus	Alibaba	OpenRouter	Z
Qwen3-235B-A22B	Alibaba	OpenRouter	Z
Llama-3.1-8B	Meta	OpenRouter	Z
Qwen-2.5-Coder-32B	Alibaba	OpenRouter	Z
<i>Inference.net models</i>			
Gemma-3-27B	Google	Inference.net	A

All models are evaluated with temperature 0.0 for reproducibility, with a maximum of 8,192 generated tokens per request. OpenRouter models are evaluated with 1 trial per (model, problem) pair under zero-shot prompting. Gemma-3-27B is evaluated with 3 trials across all five prompting strategies (Section 5.4), yielding $3 \times 12 \times 5 = 180$ total runs for the ablation study.

5.3.2 Automated Metric Computation

Each generated code sample passes through a four-stage analysis pipeline:

1. **Syntax Validation:** Python AST parsing (`ast.parse`) to verify syntactic correctness.
2. **Structural Checks:** detection of at least one `Scene` subclass and valid `from manim import *` imports; flagging of ManimGL-specific imports.
3. **Sandboxed Execution:** the code is written to a temporary file and rendered via `subprocess` with a configurable timeout (default: 60 s). Exit codes, `stderr`, and error types (`ImportError`, `AttributeError`, etc.) are captured.
4. **Static Analysis:** over 40 regex patterns (derived from the 145 documented GL→CE incompatibilities in Section 4.2) scan for version conflicts; keyword-bank-based heuristics detect the presence of required visual events and pedagogical elements.

All four metrics are computed fully automatically. Executability and VCER use AST parsing, subprocess execution, and regex-based static analysis. Alignment and Coverage are approximated heuristically via keyword and pattern matching, serving as conservative lower-bound estimates (see Section 5.2).

5.4 Prompt Engineering Strategies

Following the ManiBench Prompt Engineering Guide, we implement five prompting strategies of increasing sophistication, each evaluated in full on Gemma-3-27B with 3 trials per problem:

1. **Zero-Shot Direct.** The problem statement is provided verbatim with a system prompt specifying Manim CE as the target. Serves as the baseline across all models; Kimi-K2.5 achieves 66.7% executability under this strategy.
2. **Few-Shot Examples.** One or two working Manim CE code examples precede the target problem. Examples are kept short (20–30 lines) and domain-relevant. In our ablation (Section 6.4), few-shot is the only strategy to improve executability for Gemma-3-27B (+11.1 pp) and yields a modest coverage gain (+2.8 pp).
3. **Chain-of-Thought (CoT).** The model is instructed to first analyze visual components, event ordering, transformations, timing, and required labels before writing code. Empirically, CoT introduces a small version-conflict rate (2.8%) on Gemma-3-27B and slightly degrades coverage, suggesting the reasoning trace can lead to deprecated API references.
4. **Constraint-Based.** Explicit timing, ordering, and criticality constraints are injected (e.g., “gradient arrow must appear *before* dot moves”). Aimed at Level 3–4 problems; in practice, yields marginal changes in coverage and does not improve executability for Gemma-3-27B.
5. **Version-Conflict-Aware.** The system prompt enumerates forbidden ManimGL constructs (derived from Section 4.2), and problem-specific incompatibilities from the dataset’s `version_conflict_notes` field are appended. Eliminates version-conflict errors entirely for Gemma-3-27B (VCER = 0.0%) and achieves perfect heuristic alignment (1.0).

6 Results

6.1 Experimental Setup

Models. We evaluate six models across two API providers (Table 3). Five models are accessed via OpenRouter under zero-shot prompting with a single trial per (model, problem) pair, yielding $5 \times 12 = 60$ samples. Gemma-3-27B is evaluated via Inference.net across all five prompting strategies with 3 trials each, yielding 180 additional samples. The total evaluation corpus comprises 240 generated code samples.

Environment. All code targets Manim CE, executed in a sandboxed environment with a 60-second rendering timeout. Metrics are computed using the automated pipeline described in Section 5.3. Alignment and Coverage scores reported here are heuristic lower-bound estimates based on keyword and pattern matching; no human evaluation has been conducted in this study.

6.2 Cross-Model Comparison (Zero-Shot)

Table 4 reports aggregate results for all six models under zero-shot prompting.

Key Findings. Kimi-K2.5 achieves the highest executability (66.7%), successfully rendering 8 of 12 problems. Qwen3-235B-A22B and Gemma-3-27B produce zero version-conflict errors but struggle with executability (25.0% and 0.0%, respectively). Qwen-2.5-Coder-32B fails to render any problem, with alignment scores averaging only 0.47, indicating both structural and semantic deficiencies in its Manim code. Notably, Llama-3.1-8B achieves a perfect alignment score of 1.0 (heuristic) despite only 8.3% executability, suggesting its outputs contain the expected visual-event keywords even when the code does not execute.

Table 4: Cross-model comparison under zero-shot prompting. Executability (Exec.) is the fraction of outputs that render without error. VCER is the version-conflict error rate. Alignment (Align.) and Coverage (Cov.) are heuristic estimates. Best value in each column is **bolded**.

Model	API	Exec. \uparrow	VCER \downarrow	Align. \uparrow	Cov. \uparrow
Kimi-K2.5	OpenRouter	0.667	0.083	0.917	0.265
Qwen3.5-Plus	OpenRouter	0.333	0.085	0.917	0.226
Qwen3-235B-A22B	OpenRouter	0.250	0.000	0.993	0.251
Llama-3.1-8B	OpenRouter	0.083	0.024	1.000	0.132
Qwen-2.5-Coder	OpenRouter	0.000	0.000	0.471	0.014
Gemma-3-27B	Inference.net	0.000	0.000	0.993	0.172

6.3 Per-Problem Analysis (Zero-Shot)

Table 5 breaks down results by benchmark problem, averaged across all six models.

Table 5: Per-problem results averaged across all six models (zero-shot prompting). Diff. = difficulty level (1–5).

ID	Problem	Diff.	Exec.	VCER	Align.	Cov.
MB-001	Colliding Blocks	4	0.17	0.012	0.89	0.25
MB-002	Gradient Descent	3	0.00	0.003	0.86	0.15
MB-003	Convolution	3	0.50	0.000	0.94	0.16
MB-004	Eigenvectors	4	0.17	0.167	0.60	0.10
MB-005	Determinant	2	0.17	0.000	0.88	0.16
MB-006	CLT	3	0.33	0.000	0.90	0.18
MB-007	Medical Test	2	0.50	0.000	0.92	0.19
MB-008	Chain Rule	3	0.33	0.167	0.68	0.14
MB-009	Integration	3	0.17	0.000	0.84	0.25
MB-010	Taylor Series	4	0.17	0.000	0.92	0.24
MB-011	Hairy Ball	5	0.00	0.000	0.88	0.17
MB-012	Windmill	4	0.17	0.036	1.00	0.16

6.4 Prompting Strategy Ablation (Gemma-3-27B)

Gemma-3-27B is the only model evaluated across all five prompting strategies (3 trials each), enabling a controlled ablation study. Table 6 reports the results.

Ablation Findings.

1. **Few-shot is the only strategy that improves executability** for Gemma-3-27B, achieving 11.1% Pass@1 (4 of 36 runs render successfully), concentrated on MB-006 (CLT), MB-007 (Medical Test), and MB-008 (Chain Rule).
2. **Chain-of-thought introduces version conflicts** (VCER = 2.8%) and slightly degrades alignment (0.972 vs. 0.993). The reasoning trace appears to cause the model to reference deprecated constructs.
3. **Version-aware prompting eliminates version conflicts** and achieves perfect heuristic alignment (1.0), confirming that explicit forbidden-construct lists are effective.

Table 6: Prompting strategy ablation on Gemma-3-27B (3 trials per problem, 12 problems). Few-shot prompting yields the only non-zero executability. Δ columns report the change relative to zero-shot.

Strategy	Exec.	Δ Exec.	VCER	Align.	Cov.	Δ Cov.
Zero-Shot	0.000	—	0.000	0.993	0.172	—
Few-Shot	0.111	+11.1 pp	0.000	1.000	0.200	+2.8 pp
Chain-of-Thought	0.000	+0.0 pp	0.028	0.972	0.159	-1.3 pp
Constraint	0.000	+0.0 pp	0.000	0.995	0.171	-0.1 pp
Version-Aware	0.000	+0.0 pp	0.000	1.000	0.175	+0.3 pp

4. **Coverage remains uniformly low** (0.16–0.20) across all strategies, indicating that prompting alone cannot substantially increase the density of pedagogical elements (mathematical annotations, numeric evidence, structural clarity).

6.5 Detailed Per-Model Per-Problem Grid

Table 7 presents the full executability and coverage grid for the three highest-performing OpenRouter models and Gemma-3-27B.

Table 7: Per-model per-problem executability and coverage scores (zero-shot). Exec. values are binary (1 or 0) for single-trial OpenRouter runs; averaged over 3 trials for Gemma-3-27B. • = executable, ○ = not executable.

Problem	Kimi-K2.5		Qwen3.5+		Qwen3-235B		Gemma-3-27B	
	Ex.	Cov.	Ex.	Cov.	Ex.	Cov.	Ex.	Cov.
MB-001	•	.42	○	.27	○	.34	○	.28
MB-002	○	.21	○	.17	○	.21	○	.14
MB-003	•	.20	•	.17	•	.26	○	.17
MB-004	•	.25	○	.00	○	.13	○	.16
MB-005	•	.24	○	.22	○	.17	○	.12
MB-006	•	.25	•	.27	○	.25	○	.10
MB-007	•	.32	•	.23	○	.25	○	.13
MB-008	○	.00	•	.33	•	.26	○	.19
MB-009	○	.36	○	.38	•	.39	○	.24
MB-010	•	.33	○	.31	○	.27	○	.22
MB-011	○	.27	○	.21	○	.24	○	.18
MB-012	•	.34	○	.15	○	.25	○	.14
Mean	.67	.27	.33	.23	.25	.25	.00	.17

6.6 Key Observations

1. **Executability is the primary bottleneck.** Even the best-performing model (Kimi-K2.5) renders only 66.7% of problems. The expert-level Hairy Ball theorem (MB-011, difficulty 5) was not rendered by any model.
2. **Version conflicts are sparse but concentrated.** Two problems trigger version-conflict errors across models: MB-004 (Eigenvectors, VCER = 16.7%) and MB-008

(Chain Rule, VCER = 16.7%), both involving linear-algebra transformations that tempt models into using deprecated `ShowCreation` or ManimGL-specific `CONFIG` patterns.

3. **Heuristic alignment saturates.** Most models achieve alignment scores above 0.90 because keyword-based detection cannot distinguish whether visual events are *correctly implemented* versus merely *mentioned in comments or string literals*. This underscores the need for future human review or vision-based evaluation to produce more discriminative alignment scores.
4. **Coverage is uniformly low.** Average coverage across all models is 0.18, indicating that LLM-generated Manim code lacks pedagogical richness—mathematical annotations, numeric evidence trackers, and structural clarity elements are rarely produced.
5. **Model scale does not guarantee Manim proficiency.** Qwen3-235B-A22B (235B parameters) achieves lower executability (25.0%) than Kimi-K2.5, suggesting that domain-specific training data availability matters more than raw scale for specialized code generation.

7 Discussion

7.1 Why ManiBench Matters

Existing benchmarks (HumanEval, APPS) measure whether code produces correct *output*. MANIBENCH measures whether code produces correct *understanding*. This distinction is critical for educational tools, where a silent failure (wrong animation) is worse than a loud failure (runtime error).

7.2 Limitations and Future Work

1. **No Human Evaluation.** All metrics reported in this study are computed fully automatically. Alignment and Coverage rely on keyword-bank and regex heuristics that detect concept *presence* but cannot verify *correctness* of the rendered animation. A structured human review protocol has been designed (Section 5.2) but not yet conducted; incorporating human judgments is a high-priority next step.
2. **Heuristic Alignment Saturation.** The keyword-based alignment metric saturates near 1.0 for most models (Section 6.6), failing to distinguish correct implementations from code that merely contains relevant keywords. Future work should integrate frame-level video comparison or vision–language model grading to produce more discriminative alignment scores.
3. **Limited Model Coverage.** Due to API availability constraints, only Gemma-3-27B was evaluated across all five prompting strategies. OpenRouter models were evaluated under zero-shot prompting with a single trial. Extending to full multi-trial, multi-strategy evaluation for frontier models (GPT-4o, Claude Sonnet 4, Gemini 2.5 Pro) is a priority.
4. **Pedagogical Validation.** We do not yet validate whether animations actually teach the concept. User studies with students could address this gap.
5. **Manim API Coverage.** As Manim evolves, the benchmark should be versioned and updated accordingly. The 145 documented GL→CE incompatibilities provide a starting point for automated version-conflict detection.

6. **Scalability.** Moving from 12 to 150+ problems requires annotation infrastructure and community contribution.
7. **Reference Code Utilization.** The ~53,000 lines of analyzed reference code could enable fine-tuning studies or retrieval-augmented generation (RAG) experiments.

7.3 Broader Impact

MANIBENCH can be used to:

- evaluate LLM educational-content generation across model families and scales,
- develop better prompting strategies for animation code (our ablation shows few-shot is most effective for executability),
- identify systematic failure modes (e.g., uniformly low pedagogical coverage, version-conflict concentration on transformation-heavy problems),
- drive research into improving Manim API adoption in LLMs,
- benchmark version-aware code generation using the 145 documented GL→CE incompatibilities, and
- enable retrieval-augmented generation (RAG) experiments using the reference code analysis.

8 Conclusion

We have introduced MANIBENCH, a specialized benchmark for evaluating Manim code generation. By formalizing metrics for syntactic correctness, version compliance, visual-logic alignment, and pedagogical coverage, MANIBENCH moves beyond simple test-case evaluation to assess whether generated animations actually communicate mathematical concepts.

The 12-problem pilot dataset, backed by comprehensive reference code analysis of ~53,000 lines of original 3Blue1Brown source code and 145 documented GL→CE incompatibilities, reveals that even the best-performing model (Kimi-K2.5) renders only 66.7% of problems, while coverage of pedagogical elements averages just 0.18. Our prompting strategy ablation on Gemma-3-27B shows that few-shot examples are the only strategy to measurably improve executability (+11.1 pp), while version-aware prompting effectively eliminates version-conflict errors. The accompanying automated evaluation framework—supporting models across OpenRouter and Inference.net with five prompting strategies—enables reproducible assessment. With planned expansion to 150–200 problems and integration of vision-based alignment scoring, MANIBENCH will serve as a foundational resource for advancing LLM-driven educational content creation.

The dataset and evaluation toolkit are publicly available at <https://huggingface.co/datasets/nabin2004/ManiBench>.

References

- [1] M. Chen, J. Tworek, H. Jun, et al., “Evaluating Large Language Models Trained on Code,” *arXiv:2107.03374*, 2021. (HumanEval)
- [2] J. Austin, A. Odena, M. Nye, et al., “Program Synthesis with Large Language Models,” *arXiv:2108.07732*, 2021. (MBPP)

- [3] D. Hendrycks, S. Basart, S. Kadavath, et al., “Measuring Coding Challenge Competence with APPS,” *NeurIPS Datasets and Benchmarks*, 2021.
- [4] G. Sanderson, “3Blue1Brown,” <https://www.3blue1brown.com/>, 2015–present.
- [5] Manim Community Developers, “Manim—Mathematical Animation Framework (Community Edition),” <https://www.manim.community/>, 2020–present.
- [6] G. Sanderson, “Manim (3Blue1Brown fork),” <https://github.com/3b1b/manim>, 2015–present.
- [7] T. Brown, B. Mann, N. Ryder, et al., “Language Models are Few-Shot Learners,” *NeurIPS*, 2020.
- [8] J. Wei, X. Wang, D. Schuurmans, et al., “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” *NeurIPS*, 2022.
- [9] OpenAI, “GPT-4o Technical Report,” 2024.
- [10] Anthropic, “Claude Sonnet 4 Model Card,” 2025.
- [11] Google DeepMind, “Gemini 2.5 Pro Technical Report,” 2025.
- [12] DeepSeek, “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning,” *arXiv:2501.12948*, 2025.

A Problem Annotation Template

Each problem in MANIBENCH is annotated as a JSON object with the following structured metadata:

id: unique identifier (e.g., MB-001).

title: descriptive title.

youtube_video_id: YouTube video identifier.

category: list of task categories (e.g., `["drift-sensitive", "multi-scene"]`).

difficulty_level: integer 1–5.

domain: list of mathematical domain(s).

full_prompt: full natural-language problem statement (used as the LLM input).

raw_code_status: whether original 3B1B source code has been collected.

raw_code_path: relative path to original ManimGL source files.

reference_code_analysis: structured analysis of original code, including:

framework: source framework (e.g., `manim_g1`).

total_lines: lines of original code.

scene_classes: list of scene classes with descriptions and key methods.

visual_techniques: catalog of rendering and animation patterns.

manim_api_patterns: updaters, animation types, 3D constructs, custom classes.

required_visual_events: list of events, each with an identifier, description, weight, criticality flag, timing, and reference code location.

coverage_requirements: list of required pedagogical elements.

version_conflict_notes: GL→CE incompatibilities specific to the problem.

success_criteria: minimum thresholds for executability, alignment, coverage, and version-conflict error rate.

common_failure_modes: known LLM failure patterns with severity tags.

The full dataset is available in JSON format at <https://huggingface.co/datasets/nabin2004/ManiBench>.