Automated Neurosymbolic World Modeling via Law Evolution and Neural Correction

Nguyen Viet Tuan Kiet[1⋆], Duong Quoc Chinh[1⋆]
Nguyen Huu Duc[1⋆⋆], Bui Dinh Pham[1⋆⋆], Tran Cong Dao[2⋆⋆], Huynh Thi Thanh Binh[1†]
[1]Hanoi University of Science and Technology, Vietnam [2]University of Sydney, Australia
⋆Co-first authors ⋆⋆Co-second authors †Corresponding author: binhht@soict.hust.edu.vn

***Abstract*—We introduce Law Evolution and Neural Correction (LEAN), a framework for Automated Neurosymbolic World Modeling (ANWM) in continuous-control domains. Prior approaches to code-based symbolic world models have primarily focused on discrete environments, where state transitions can be represented as symbolic object updates or finite rule applications. In contrast, robotics and MuJoCo-style control tasks require modeling high-dimensional continuous dynamics, contact-rich interactions, and action-dependent physical effects. LEAN addresses this gap by automatically evolving executable symbolic dynamics programs from transition data and combining them with neural correction modules that refine their predictions where symbolic laws are incomplete. Our method searches over compact Python laws using LLM-guided mutation, crossover, validation, and likelihood-based selection, producing interpretable symbolic components over continuous state variables such as positions and velocities. These evolved laws are then fitted as a weighted product of experts and enhanced with neural residual or trust-region corrections for accurate rollout prediction and model-predictive control. Experiments across continuous-control robotic benchmarks show that LEAN enables automated construction of world models that retain symbolic interpretability while achieving the flexibility needed for robotics planning.**

I. Introduction

World models [1, 2] provide a compact predictive interface between an agent and its environment, enabling agents to learn, imagine, and plan without repeatedly querying the real system. They are central to model-based reinforcement learning [3], where learned dynamics support trajectory optimization, policy improvement, and latent-space planning. Neural world models such as PETS [4], PlaNet [5], Dreamer [6], and TD-MPC [7] have demonstrated strong performance in continuous-control domains, including MuJoCo [8] robotics benchmarks. However, their dense representations are difficult to inspect, edit, or reuse compositionally, which is limiting in robotics where physical structure, safety constraints, and failure diagnosis often matter alongside predictive accuracy [9].

A complementary line of work studies symbolic and neurosymbolic world models. WorldCloner [10] represents dynamics through executable symbolic structure, while PoE [11] and OneLife [12] show that symbolic laws can be inferred from experience and combined probabilistically. These approaches yield interpretable transition models that can be inspected and composed. Yet existing code-based symbolic world models have primarily focused on discrete environments, where objects, attributes, and state transitions are finite symbolic updates. Continuous-control robotics is qualitatively different: states and actions are high-dimensional and real-valued, dynamics are smooth but nonlinear, and useful laws must predict positions, velocities, contact effects, and action-dependent accelerations. Thus, useful laws should remain executable while exposing uncertainty over real-valued predictions.

Recent progress in program synthesis and LLM-guided search suggests a path toward automating symbolic model construction. DreamCoder [13] learns reusable symbolic programs with neural guidance, FunSearch [14] couples large language models with evolutionary search and automated evaluators, and Code as Policies [15] demonstrates executable robot policy code. These works show that code is expressive, compositional, and human-readable. However, program search alone is brittle for continuous dynamics: compact laws may capture dominant physical trends, but residual errors can accumulate over multi-step rollouts and degrade planning. Conversely, purely neural correction can improve accuracy but may obscure the symbolic structure that makes the model interpretable.

We introduce *Law Evolution and Neural Correction* (LEAN), a framework for *Automated Neurosymbolic World Modeling* (ANWM) in continuous-control domains. LEAN evolves executable Python dynamics laws from transition data, fits them as a weighted product of experts, and converts their discretized predictions into continuous predictive centers and uncertainties. To handle the gap between compact laws and complex robot dynamics, LEAN augments the symbolic model with gated residual or trust-region neural corrections. The resulting world model preserves inspectable symbolic dynamics while providing the flexibility needed for accurate rollout prediction and model-predictive control. This design targets planning, not only one-step prediction.

Our work makes three contributions. First, we formulate ANWM for continuous-control robotics, extending code-based symbolic world modeling beyond the discrete settings studied by WorldCloner [10], PoE [11], and OneLife [12]. Second, we develop LEAN, which combines executable symbolic law evolution, product-of-experts fitting, and adaptive neural correction. Third, we evaluate LEAN on MuJoCo [8] benchmarks, testing whether evolved symbolic laws plus neural correction can yield world models that are accurate enough for rollout prediction and model-predictive control while remaining more interpretable than purely neural alternatives such as PETS [4], Dreamer [6], and TD-MPC [7].

II. Preliminaries

A. Continuous-Control World Models

We consider a continuous-control Markov decision process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, the standard formalism for reinforcement learning and model-based control [3]. Here, $\mathcal{S} \subseteq \mathbb{R}^{d_s}$ is a continuous state space, $\mathcal{A} \subseteq \mathbb{R}^{d_a}$ is a continuous action space, $P(s' \mid s, a)$ is the transition kernel, $R(s, a, s')$ is the reward function, and $\gamma \in [0, 1)$ is the discount factor. In MuJoCo [8], the state typically contains generalized positions and velocities, written as $s_t = [q_t, \dot{q}_t]$, while the action a_t corresponds to continuous actuator commands.

A world model [1, 2] is a learned approximation of the transition dynamics. Given a dataset of transitions $\mathcal{D} = \{(s_t, a_t, s_{t+1})\}_{t=1}^{N}$, a deterministic world model predicts the next state as $\hat{s}_{t+1} = f_\theta(s_t, a_t)$. Neural world models used in model-based reinforcement learning, including PETS [4], PlaNet [5], Dreamer [6], and TD-MPC [7], are commonly trained by minimizing one-step prediction error:

$$\mathcal{L}_{\text{dyn}}(\theta) = \frac{1}{N} \sum_{t=1}^{N} \| f_\theta(s_t, a_t) - s_{t+1} \|_2^2 . \tag{1}$$

B. Model-Based Planning

Model-based reinforcement learning uses a learned dynamics model to evaluate candidate action sequences before executing actions in the real environment [3]. Given a planning horizon H, a planner searches for an action sequence $a_{t:t+H-1}$ that maximizes predicted return under the learned model:

$$a^\star_{t:t+H-1} = \arg\max_{a_{t:t+H-1}} \sum_{k=0}^{H-1} R(\hat{s}_{t+k}, a_{t+k}, \hat{s}_{t+k+1}), \tag{2}$$

where the predicted states are generated recursively by $\hat{s}_t = s_t$ and $\hat{s}_{t+k+1} = f_\theta(\hat{s}_{t+k}, a_{t+k})$.

In model-predictive control, only the first action $a^\star_t$ is executed, and planning is repeated after observing the next state. This planning interface is shared by many neural model-based methods, including PETS [4] and TD-MPC [7]. Its performance depends on the rollout accuracy of the learned model, since one-step errors can accumulate over the planning horizon. We denote the H-step rollout error by

$$\epsilon_H = \| \hat{s}_{t+H} - s_{t+H} \|_2 . \tag{3}$$

C. Code-Based Symbolic World Models

Code-based symbolic world models represent transition dynamics using executable rules or programs. WorldCloner [10] studies neuro-symbolic world modeling with executable symbolic structure, while PoE [11] and OneLife [12] combine symbolic laws probabilistically through a product-of-experts formulation.

A symbolic law can be written as $\ell_i = (c_i, e_i)$, where $c_i(s, a) \in \{0, 1\}$ is a precondition deciding whether the law applies, and e_i is an effect that predicts some next-state attributes. For a state attribute indexed by j, an active law produces a distribution $p_i(b \mid s, a, j)$ over possible next values. In discrete symbolic environments, this distribution is naturally defined over a finite value set. For continuous states, one may discretize each dimension into bins $\mathcal{B}_j$, with a map $\psi_j : \mathcal{B}_j \to \mathbb{R}$ from bins to continuous representative values.

Following the product-of-experts view used in PoE [11] and OneLife [12], multiple active laws can be combined by adding their weighted log-scores:

$$\log \tilde{p}_{\text{sym}}(b \mid s, a, j) = \sum_{i \in \mathcal{I}_j(s,a)} w_i \log p_i(b \mid s, a, j), \tag{4}$$

where $\mathcal{I}_j(s, a)$ denotes the set of active laws that predict attribute j, and $w_i \geq 0$ is the learned reliability weight of law ℓ_i. The normalized symbolic predictive distribution is

$$p_{\text{sym}}(b \mid s, a, j) = \frac{\exp\left(\log \tilde{p}_{\text{sym}}(b \mid s, a, j)\right)}{\sum_{b' \in \mathcal{B}_j} \exp\left(\log \tilde{p}_{\text{sym}}(b' \mid s, a, j)\right)}. \tag{5}$$

The law weights can be fitted by maximum likelihood on observed transitions:

$$\begin{aligned} \mathcal{L}_{\text{sym}}(w) = &- \sum_{(s_t, a_t, s_{t+1}) \in \mathcal{D}} \sum_{j=1}^{d_s} \log p_{\text{sym}}(b_j(s_{t+1}) \mid s_t, a_t, j) \\ &+ \lambda \| w \|_1 . \end{aligned} \tag{6}$$

The symbolic predictive distribution can be summarized by a continuous mean and variance:

$$\begin{aligned} \mu_{\text{sym},j}(s, a) &= \sum_{b \in \mathcal{B}_j} p_{\text{sym}}(b \mid s, a, j) \psi_j(b), \\ \sigma^2_{\text{sym},j}(s, a) &= \sum_{b \in \mathcal{B}_j} p_{\text{sym}}(b \mid s, a, j) \left(\psi_j(b) - \mu_{\text{sym},j}(s, a)\right)^2. \end{aligned} \tag{7}$$

This formulation provides an interpretable transition model: each prediction can be traced to a set of active executable laws and their fitted weights. However, prior code-based symbolic world models such as WorldCloner [10], PoE [11], and OneLife [12] are primarily developed for discrete symbolic settings, motivating extensions to continuous-control domains.

III. Methodology

We propose LEAN for automated neurosymbolic world modeling in continuous-control domains. LEAN has two stages: it first searches for executable symbolic laws that explain transition data, then learns a neural correction model around the fitted symbolic prediction. The resulting model can be rolled out inside a model-predictive controller.

A. Symbolic Law Evolution

Given a transition dataset $\mathcal{D} = \{(s_t, a_t, s_{t+1})\}_{t=1}^{N}$, LEAN searches for a library of executable symbolic laws $\Lambda = \{\ell_i\}_{i=1}^{M}$. Each law $\ell_i = (c_i, e_i)$ consists of a precondition c_i and an effect e_i. The effect writes a distribution over a small subset of next-state dimensions, rather than predicting the full transition at once.

LEAN maintains a population of candidate laws. For a selected transition $\tau = (s, a, s')$, an LLM proposes a Python law conditioned on the current state, action, next state, dimension semantics, and optional feedback from previous symbolic errors. Each generated law is parsed, compiled in a restricted

namespace, and validated by executing it on held-out transitions. Invalid programs are discarded.

Each valid candidate ℓ_i is scored by combining predictive likelihood, coverage, and a complexity penalty. Let $J_i(s,a)$ be the set of state dimensions predicted by ℓ_i, and let $b_j(s')$ be the discretized bin of the observed next-state dimension j. Writing $\tau = (s,a,s')$, we score a law by

$$\begin{aligned} S(\ell_i) = & \frac{1}{C_i} \sum_{\tau \in \mathcal{V}} \sum_{j \in J_i(s,a)} \log p_i(b_j(s') \mid s,a,j) \\ & + \alpha \frac{C_i}{|\mathcal{V}| d_s} - \beta K_i, \end{aligned} \tag{8}$$

where $\mathcal{V}$ is a validation set, $C_i = \sum_{\tau=(s,a,s') \in \mathcal{V}} |J_i(s,a)|$ is the number of predicted attributes, and K_i is a code-length penalty.

The search then improves the population through mutation and crossover. Mutation modifies one law by changing preconditions, target dimensions, coefficients, or coupling terms. Crossover combines reusable pieces from two high-scoring parent laws, such as taking one parent's precondition and another parent's effect. After each generation, candidates are deduplicated by behavior and ranked by $S(\ell_i)$. This produces a compact library of executable symbolic dynamics programs.

B. Fitted Symbolic Dynamics

After law evolution, LEAN fits the law library Λ using the product-of-experts form in (4) and the normalized symbolic distribution in (5). The weights $w_i \geq 0$ are optimized with the maximum-likelihood objective in (6). The resulting symbolic model is summarized by the continuous predictive center and variance in (7); in this section, we write them more compactly as $\mu_j(s,a)$ and $v_j(s,a)$.

We write $\mu(s,a) = [\mu_1(s,a), \ldots, \mu_{d_s}(s,a)]$ and $v(s,a) = [v_1(s,a), \ldots, v_{d_s}(s,a)]$. Besides μ and v, the model exposes symbolic features such as law weights, active-law indicators, and coverage masks. We collect these into a feature vector $\phi(s,a)$, which is passed to the neural correction module. For compact notation at time t, let $\mu_t = \mu(s_t,a_t)$, $v_t = v(s_t,a_t)$, $\phi_t = \phi(s_t,a_t)$, and $y_t = s_{t+1}$.

C. Neural Correction for Continuous Dynamics

The evolved symbolic laws are interpretable but intentionally compact, so they may miss contact effects, nonlinear couplings, or small continuous residuals. LEAN therefore learns a neural correction around the symbolic prediction $\mu(s,a)$. We study two parameterizations.

a) Gated Residual Correction (GR): A residual network predicts $r_\theta(\phi) \in \mathbb{R}^{d_s}$, and a gate network predicts $g_\theta(\phi) \in [0,1]^{d_s}$. The corrected next-state prediction is

$$\begin{aligned} r_t &= r_\theta(\phi_t), \qquad g_t = g_\theta(\phi_t), \\ \hat{s}^{\mathrm{GR}}_{t+1} &= \mu_t + g_t \odot r_t. \end{aligned} \tag{9}$$

Training is performed in two phases. First, the residual is warmed up with the gate forced open. Then the gate is trained to open only when the residual improves over the symbolic prediction. The gated objective is

$$\begin{aligned} \mathcal{J}_g = & \|\mu_t + g_t \odot r_t - y_t\|_2^2 + \lambda_1 B(g_t, \bar{g}_t) \\ & + \lambda_2 \|g_t\|_1 + \lambda_3 \|r_t\|_2^2, \end{aligned} \tag{10}$$

where $\bar{g}_t$ is a target gate indicating whether the full residual improves prediction error for each state group, and B is binary cross-entropy.

b) Trust-Region Correction (TR): Instead of learning an unrestricted residual, the neural model predicts a normalized offset $z_\theta(\phi) \in [-1,1]^{d_s}$ inside a symbolic uncertainty region. The region width is derived from the symbolic variance:

$$h_j(s,a) = \mathrm{clip}\left(\kappa \sqrt{v_j(s,a)}, h_{\min}, h_{\max}\right). \tag{11}$$

To handle cases where the symbolic region is too narrow, the model also predicts a nonnegative escape term $e_\theta(\phi) \geq 0$. Let $h_t = h(s_t,a_t)$. The prediction is

$$\begin{aligned} e_t &= e_\theta(\phi_t), \qquad z_t = z_\theta(\phi_t), \\ \hat{s}^{\mathrm{TR}}_{t+1} &= \mu_t + (h_t + e_t) \odot z_t. \end{aligned} \tag{12}$$

The trust-region objective penalizes prediction error, mismatch in normalized coordinates, and excessive escape. Let

$$\bar{z}_t = \mathrm{clip}\left(\frac{y_t - \mu_t}{h_t}, -1, 1\right). \tag{13}$$

Then the objective is

$$\begin{aligned} \mathcal{J}_z = & \left\|\hat{s}^{\mathrm{TR}}_{t+1} - y_t\right\|_2^2 + \lambda_4 \rho(z_t, \bar{z}_t) \\ & + \lambda_5 \left\|e_t\right\|_1. \end{aligned} \tag{14}$$

where ρ is a smooth L_1 loss. This correction keeps predictions near the symbolic dynamics unless the data supports moving outside the symbolic trust region.

The final learned world model is rolled out recursively during planning:

$$\hat{s}_t = s_t, \qquad \hat{s}_{t+k+1} = f(\hat{s}_{t+k}, a_{t+k}), \tag{15}$$

where f denotes either corrected model above. A model-predictive controller selects the first action of the best predicted sequence and replans after observing the next state.

IV. Experiments

We evaluate whether automated symbolic law evolution and neural correction improve both world-model accuracy and downstream planning in continuous-control environments. We use six MuJoCo tasks [8]: Swimmer, Reacher, Hopper, Pusher, Walker2D, and Ant. These tasks cover low-dimensional smooth locomotion, goal-reaching, object interaction, and high-dimensional articulated control.

TABLE I
WORLD-MODEL ROLLOUT ACCURACY ON MUJOCO TASKS.

Method	Swimmer		Reacher		Hopper		Pusher		Walker2D		Ant		Avg Rank
	$R^2@1$	$R^2@10$	$R^2@1$	$R^2@10$	$R^2@1$	$R^2@10$	$R^2@1$	$R^2@10$	$R^2@1$	$R^2@10$	$R^2@1$	$R^2@10$	
OneLife	0.8246	0.1230	0.9582	**0.6366**	0.8896	0.4544	0.9499	**0.6519**	0.7982	-0.0955	0.4091	-0.6183	4.67
Dreamer V3	0.9924	0.9416	**0.9924**	0.1399	0.9860	0.7550	**0.9747**	0.3067	**0.9750**	**0.9747**	0.7618	-1.9928	2.75
PETS dynamics	0.9920	0.8544	0.9884	-0.3424	**0.9901**	**0.9733**	0.5323	-0.2591	0.9720	0.6946	0.6906	0.0423	3.58
SWM-TR	0.9670	0.5711	0.9529	-0.0610	0.9684	0.2427	0.9490	0.5073	0.9268	-0.1616	0.6090	-0.0931	4.92
NSWM-GR	**0.9952**	0.9509	0.9834	0.1542	0.9831	0.5844	0.9597	0.4829	**0.9779**	0.5727	0.8274	0.1010	**2.50**
NSWM-TR	0.9942	**0.9525**	0.9911	-0.0227	0.9827	0.9615	0.9681	0.4773	0.9489	0.3122	**0.8340**	**0.1337**	2.58

A. Experimental Setup

We evaluate LEAN on six MuJoCo continuous-control tasks [8]: Swimmer, Reacher, Hopper, Pusher, Walker2D, and Ant. For controlled comparison, we reproduce neural and symbolic baselines with compact architectures whose parameter counts match our correction networks, rather than large task-specialized implementations. PETS dynamics [4] is a small probabilistic ensemble, Dreamer V3 [6] a compact latent dynamics baseline, and OneLife [12] a code-based symbolic baseline adapted to continuous states through discretization. SWM-TR is a symbolic trust-region baseline without neural correction, while NSWM-GR and NSWM-TR are LEAN variants with gated residual and trust-region neural correction.

LEAN adds a modest symbolic search stage before neural training. In each task, the search maintains a small population of candidate laws, uses LLM-guided mutation and crossover for a few generations, validates executable programs on held-out transitions, and fits the retained laws as a weighted product of experts. The subsequent neural correction networks are intentionally small and trained with the same transition budget as the baselines. We report rollout $R^2@H$ for world-model accuracy and true MuJoCo episode return for CEM-MPC planning. Average rank is computed across tasks and metrics, with lower values indicating better overall performance.

B. World-Model Rollout Accuracy

Table I reports rollout accuracy across MuJoCo tasks. The results show that no single baseline dominates across all domains and horizons. Dreamer V3 performs strongly on several one-step predictions and achieves the best Walker2D long-horizon accuracy, while PETS dynamics is particularly strong on Hopper. However, both neural baselines can degrade substantially on harder long-horizon rollouts, as seen on Reacher, Pusher, and Ant.

The symbolic baselines behave differently. OneLife achieves competitive short-horizon accuracy on some lower-dimensional tasks, but its long-horizon performance often drops, especially on Ant. This supports the hypothesis that symbolic world models designed for discrete environments do not directly transfer to continuous-control settings without additional mechanisms for handling real-valued dynamics. SWM-TR improves over purely discrete symbolic modeling in some cases, but remains less competitive than the neural-corrected variants.

NSWM-GR and NSWM-TR obtain the best average ranks. NSWM-GR achieves the strongest overall rollout rank, with particularly high accuracy on Swimmer and Ant. NSWM-TR is close behind and gives the best long-horizon result on Swimmer and Ant, while also remaining competitive on Hopper. These results suggest that evolved symbolic laws provide a useful dynamics prior, but neural correction is important for robust continuous-control prediction.

TABLE II
CEM-MPC PLANNING RETURN ON MUJOCO TASKS.

Method	Swimmer	Reacher	Hopper	Pusher	Walker2D	Ant	Avg Rank
OneLife	13.7421	-8.9134	214.5872	-17.6240	188.4063	103.5818	5.83
Dreamer V3	22.3187	-5.8742	691.3350	-11.9048	**910.7724**	338.2195	2.67
PETS dynamics	19.6045	-7.4361	548.9027	-13.7716	653.4809	281.6640	4.17
SWM-TR	16.9823	-8.1025	302.1478	-15.2843	274.9921	146.7334	4.92
NSWM-GR	25.7316	**-5.2149**	627.4588	-6.8821	842.5167	487.3089	2.08
NSWM-TR	**27.0842**	-5.6930	**705.1194**	**-4.9627**	781.2046	**526.7412**	**1.33**

C. CEM-MPC Planning Return

Table II reports true environment return when the learned models are used inside CEM-MPC. Unlike the rollout metric, this evaluation measures whether model errors affect action selection and downstream control. The results show that improved world-model accuracy generally translates into stronger planning performance, but the relationship is not perfectly monotonic because planning depends on task reward geometry and model errors along selected trajectories.

NSWM-TR achieves the best average rank and the best return on Swimmer, Hopper, Pusher, and Ant. NSWM-GR achieves the best return on Reacher and remains competitive on the other tasks. Dreamer V3 remains a strong baseline, especially on Walker2D, but performs worse than the NSWM variants on several tasks where interpretable symbolic dynamics and neural correction appear to provide useful structure for planning. PETS dynamics is competitive but less consistent across tasks, while OneLife and SWM-TR lag behind the neural-corrected variants.

Overall, the experiments indicate that LEAN improves automated world modeling in two complementary ways. Evolved symbolic laws provide interpretable structure and useful physical priors, while neural correction improves continuous rollout accuracy and downstream control. The gap between SWM-TR and NSWM-TR further suggests that symbolic trust-region structure alone is not sufficient; adaptive neural correction is necessary for high-quality planning in continuous-control robotics.

Author Contributions

Nguyen Viet Tuan Kiet led the project, formulated the research problem, coordinated the overall technical direction, and took primary responsibility for manuscript writing. Duong Quoc Chinh contributed to the methodology design and implementation of the proposed framework. Bui Dinh Pham, Nguyen Huu Duc, and Tran Cong Dao contributed to completing the experimental evaluation, analyzing results, and critically reviewing the technical ideas. Huynh Thi Thanh Binh served as the faculty advisor, supervised the project, provided scientific guidance, refined the research direction, and reviewed the manuscript.

References

[1] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2(3):440, 2018.

[2] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/2de5d16682c3c35007e4e92982f1a2ba-Paper.pdf.

[3] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based reinforcement learning: A survey. *Foundations and Trends in Machine Learning*, 16(1):1–118, 2023.

[4] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31, 2018.

[5] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

[6] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[7] Nicklas A Hansen, Hao Su, and Xiaolong Wang. Temporal difference learning for model predictive control. In *International Conference on Machine Learning*, pages 8387–8406. PMLR, 2022.

[8] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.

[9] Claire Glanois, Paul Weng, Matthieu Zimmer, Dong Li, Tianpei Yang, Jianye Hao, and Wulong Liu. A survey on interpretable reinforcement learning. *Machine Learning*, 113(8):5847–5890, 2024.

[10] Jonathan C Balloch, Zhiyu Lin, Xiangyu Peng, Mustafa Hussain, Aarun Srinivas, Robert Wright, Julia M Kim, and Mark O Riedl. Neuro-symbolic world models for adapting to open world novelty. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 2848–2850, 2023.

[11] Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. Poe-world: Compositional world modeling with products of programmatic experts. *Advances in Neural Information Processing Systems*, 38:26609–26638, 2026.

[12] Zaid Khan, Archiki Prasad, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=UQ36IrVCw2.

[13] Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251): 20220050, 2023.

[14] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2023. doi: 10.1038/s41586-023-06924-6.

[15] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE, 2023.

APPENDIX

We provide the main experimental and implementation details used in our MuJoCo experiments. We omit low-level software defaults and report only hyperparameters that materially affect data collection, symbolic law evolution, neural correction, or planning.

A. Evaluation Protocol

All methods are evaluated on six MuJoCo continuous-control tasks [8]: Swimmer, Reacher, Hopper, Pusher, Walker2D, and Ant. For each task, we collect transition data from the true simulator using smooth random policies. World-model accuracy is measured by multi-step rollout $R^2@H$, while planning performance is measured by true MuJoCo episode return after executing actions selected by CEM-MPC. The planner uses the learned world model only for imagined rollouts; reported returns are always computed by the real environment.

TABLE III
EVALUATION PROTOCOL.

Setting	Value
Training transitions per task	5,000
Evaluation transitions per scenario	200
Evaluation scenarios	3 action scales
Rollout horizons	$1, 3, 5, 10, 25$
Planning method	CEM-MPC
Planning episodes	10
Episode length	1,000
CEM horizon	8
CEM candidates	256
CEM iterations	5

For planning, we use the same CEM-MPC configuration for all learned models. Candidate action sequences are sampled, smoothed, rolled out in the learned model, and ranked by the task reward proxy computed from predicted states. The first action of the best sequence is executed in MuJoCo, after which the planner replans from the observed state.

B. Symbolic Law Evolution

LEAN uses an LLM-guided evolutionary search to construct executable symbolic dynamics laws. We use `gpt-4o-mini` as the code-generation model. Each synthesis round selects high-error or high-motion transitions, asks the model to propose compact Python laws, validates the generated programs, and scores valid laws by predictive log-likelihood, coverage, and code complexity. The best laws are retained and improved through mutation and crossover.

TABLE IV
MAIN SYMBOLIC LAW EVOLUTION SETTINGS.

Setting	Value
LLM	`gpt-4o-mini`
Synthesis rounds	3
Population size	14
Generations per round	3
Elite count	6
Mutation candidates	6
Crossover candidates	4
Validation transitions	48
Error pool size	48

After law evolution, retained laws are fitted as the weighted product of experts in (4)–(5), using the maximum-likelihood objective in (6). We use 8 fitting iterations with batch size 1,000 and learning rate 0.1. The fitted symbolic model produces the predictive center and per-dimension variance in (7), which are passed to the neural correction module.

C. Neural Correction and Planning Hyperparameters

Both neural correction variants use small multilayer perceptrons with hidden dimension 256 and 3 layers. NSWM-GR trains a residual network followed by a gate network, while NSWM-TR trains a trust-region correction parameterized by a normalized offset and an escape term. We use AdamW with learning rate 3×10^{-4} for both variants.

The neural correction networks are intentionally compact so that improvements cannot be attributed simply to using substantially larger models than the baselines. All baselines are reproduced with similarly small architectures and trained under the same transition budget whenever applicable.

TABLE V
MAIN NEURAL CORRECTION HYPERPARAMETERS.

Setting	NSWM-GR	NSWM-TR
Hidden dimension	256	256
Layers	3	3
Batch size	256	256
Learning rate	3×10^{-4}	3×10^{-4}
Training epochs	120 warmup + 240 gate	300
Rollout loss weight	0.02	0.02
Gate benefit weight	0.25	–
Gate budget weight	10^{-3}	–
Residual penalty	10^{-5}	–
Trust-region scale κ	–	2.0
Escape penalty	–	10^{-3}

D. Symbolic Law Evolution Algorithm

Algorithm 1 summarizes the symbolic code-generation procedure used by LEAN. The search is organized as an outer repair loop and an inner evolutionary loop. At the beginning of each repair round, LEAN selects transitions that are either high-motion or poorly explained by the current symbolic model. These transitions define synthesis niches for the LLM. For each niche, the LLM proposes executable Python laws. Candidate laws are parsed, compiled in a restricted namespace, validated on held-out transitions, and scored by the likelihood, coverage, and complexity terms in (8).

The population is then improved through mutation and crossover. Mutation asks the LLM to modify a high-scoring parent law by adjusting preconditions, target dimensions, numerical coefficients, damping terms, or action couplings. Crossover asks the LLM to combine reusable pieces from two parent laws, such as one law's precondition and another law's effect. After each generation, behaviorally duplicate laws are removed and the top-scoring laws are kept. Between repair rounds, the current law library is temporarily fitted as a weighted product of experts; its largest prediction errors are used as feedback for the next round. This feedback makes the search increasingly focused on dynamics that the current symbolic model fails to explain.

E. Mathematical Analysis of LEAN

This section expands the mathematical structure of LEAN. We focus on how the fitted symbolic model, gated residual correction, and trust-region correction interact. Throughout the appendix, let $y = s_{t+1}$, let $\hat{y} = \hat{s}_{t+1}$, let $\mu = \mu(s_t, a_t)$ be the continuous predictive center from (7), and let $v = v(s_t, a_t)$ be its per-dimension variance. The symbolic prediction error is

$$\delta = y - \mu. \tag{16}$$

A purely symbolic model is accurate when δ is small and structured errors are rare. In continuous-control tasks, however, compact symbolic laws may explain dominant trends such as velocity damping or position integration, while missing small nonlinear effects. LEAN therefore treats μ as an interpretable prior and learns a correction around it:

$$\hat{y} = \mu + \Delta_\theta(\phi), \tag{17}$$

where $\phi = \phi(s_t, a_t)$ is the symbolic feature vector described in the main text, including μ, v, active-law indicators, coverage masks, and learned law weights. The correction term Δ_θ is constrained differently in NSWM-GR and NSWM-TR.

F. Gated Residual Correction

NSWM-GR models the residual error directly. A residual network predicts $r_\theta(\phi) \in \mathbb{R}^{d_s}$, while a gate network predicts $g_\theta(\phi) \in [0,1]^{d_s}$. In the time-index-free notation of (9), the prediction is

$$\hat{y}^{\mathrm{GR}} = \mu + g_\theta(\phi) \odot r_\theta(\phi). \tag{18}$$

This parameterization can be interpreted as a dimension-wise interpolation between the symbolic model and a corrected model. When $g_{\theta,j}(\phi) = 0$, dimension j uses the symbolic prediction μ_j. When $g_{\theta,j}(\phi) = 1$, dimension j uses the full neural residual correction $\mu_j + r_{\theta,j}(\phi)$. Intermediate values allow partial correction:

$$\hat{y}_j^{\mathrm{GR}} = (1 - g_{\theta,j}(\phi))\mu_j + g_{\theta,j}(\phi)(\mu_j + r_{\theta,j}(\phi)). \tag{19}$$

Thus, the gate controls how much the model should trust the symbolic dynamics versus the neural residual. If the symbolic law mixture already predicts dimension j well, the optimal gate should be close to zero. If the residual consistently reduces error, the gate should open.

Algorithm 1 Symbolic Law Evolution

Require: Transition dataset $\mathcal{D}$, validation set $\mathcal{V}$, LLM G, repair rounds R_{rep}, generations K, population cap M_{pop}
Ensure: Symbolic law library Λ

```
Initialize law library Λ ← ∅
for r = 1, ..., R_rep do
    if Λ = ∅ then
        Select synthesis transitions Q from high-motion transitions in D
        Set feedback F ← ∅
    else
        Fit temporary law weights w for Λ on D
        Select high-error transitions Q under the fitted symbolic model
        Construct symbolic error feedback F for transitions in Q
    end if
    Initialize candidate population C ← ∅
    for each transition τ ∈ Q do
        Build prompt from τ, task semantics, seed laws, and feedback F(τ)
        Generate candidate laws {ℓ} ← G(prompt)
        for each generated law ℓ do
            if ℓ parses, compiles, and validates on V then
                Score ℓ by S(ℓ)
                Add ℓ to C
            end if
        end for
        if |C| ≥ M_pop then
            break
        end if
    end for
    Deduplicate C by behavior signature and keep top M_pop laws
    for generation = 1, ..., K do
        Let E be the elite laws in C
        Initialize children U ← ∅
        for each selected parent ℓ ∈ E do
            Generate mutated child laws from ℓ using G
            Validate and score each child; add valid children to U
        end for
        for each selected parent pair (ℓ_a, ℓ_b) ∈ E × E do
            Generate crossover child laws from (ℓ_a, ℓ_b) using G
            Validate and score each child; add valid children to U
        end for
        C ← top M_pop laws from C ∪ U after deduplication
    end for
    Add the top laws from C to Λ, ignoring exact source duplicates
end for
return Λ
```

The ideal gate decision can be understood by comparing symbolic and residual-corrected errors. Define the symbolic squared error and the full-residual squared error for dimension j:

$$E_j^0 = (y_j - \mu_j)^2, \qquad E_j^1 = (y_j - \mu_j - r_{\theta,j}(\phi))^2. \tag{20}$$

The residual is useful when $E_j^1 < E_j^0$. A dimension-level soft target for the gate is therefore

$$\bar{g}_j = \text{sigmoid}\left(\frac{E_j^0 - E_j^1 - m_g}{T_g}\right), \tag{21}$$

where m_g is a small margin and T_g is a gate temperature. In the implementation, the same comparison can be averaged over

predefined state groups to form the target gate $\bar{g}_t$ used in (10). This target encourages the gate to open only when the residual improves over the symbolic prediction by a nontrivial amount.

The training objective combines prediction loss, gate supervision, gate sparsity, and residual regularization:

$$\mathcal{J}_g = \|\mu + g_\theta(\phi) \odot r_\theta(\phi) - y\|_2^2 + \lambda_1 B(g_\theta(\phi), \bar{g}) + \lambda_2 \|g_\theta(\phi)\|_1 + \lambda_3 \|r_\theta(\phi)\|_2^2. \tag{22}$$

The first term enforces predictive accuracy. The second term teaches the gate to match the empirical benefit of the residual. The third term discourages unnecessary neural correction, preserving the symbolic model when possible. The fourth term prevents large residuals from dominating the symbolic prior.

The two-phase training procedure follows naturally from this objective. In the warmup phase, the gate is forced open, so the residual network learns an approximate residual:

$$\min_\theta \|\mu + r_\theta(\phi) - y\|_2^2. \tag{23}$$

Since $y - \mu = \delta$, the optimal residual under squared loss is $r_\theta(\phi) \approx \delta$. In the gate-training phase, the model learns when this residual should actually be used. This separation avoids a failure mode where an untrained residual causes the gate to close too early.

The advantage of NSWM-GR is flexibility. It can correct systematic symbolic errors even when those errors lie far outside the symbolic model's uncertainty estimate. The downside is that the residual is not explicitly constrained to remain near the symbolic prediction. If the gate opens too often, the model may become less interpretable because predictions depend heavily on unconstrained neural corrections.

G. Trust-Region Neural Correction

NSWM-TR uses the symbolic model's uncertainty to constrain neural correction. Instead of predicting an arbitrary residual, it defines a trust region around the symbolic prediction. Consistent with (11), for each dimension j the half-width is

$$h_j = \text{clip}\left(\kappa\sqrt{v_j}, h_{\min}, h_{\max}\right). \tag{24}$$

The interval $[\mu_j - h_j, \mu_j + h_j]$ represents the region where the symbolic model considers plausible next values.

The neural network predicts a normalized offset $z_\theta(\phi) \in [-1, 1]^{d_s}$. Without any escape term, the prediction would be

$$\hat{y}^{\text{TR}} = \mu + h \odot z_\theta(\phi). \tag{25}$$

This guarantees that each dimension remains inside its symbolic trust region:

$$|\hat{y}_j^{\text{TR}} - \mu_j| \le h_j. \tag{26}$$

However, symbolic uncertainty may be miscalibrated. A compact symbolic law library may assign a narrow variance even when the correct next state lies outside the trust region. To handle this, NSWM-TR adds a nonnegative escape term $e_\theta(\phi) \in \mathbb{R}_{\ge 0}^{d_s}$, yielding the prediction in (12):

$$\hat{y}^{\text{TR}} = \mu + (h + e_\theta(\phi)) \odot z_\theta(\phi). \tag{27}$$

The escape term expands the trust region only when needed. Since $e_\theta \ge 0$, the symbolic region is never shrunk by the neural model; it is either used as given or enlarged. The model is penalized for large escape values, so it prefers to stay within the symbolic uncertainty region unless the data supports leaving it.

The unclipped normalized target coordinate is

$$z^\star = \frac{y - \mu}{h}. \tag{28}$$

If $|z_j^\star| \le 1$, then the target lies inside the symbolic trust region for dimension j. If $|z_j^\star| > 1$, then the symbolic interval is too narrow. As in (13), the model trains the normalized offset against a clipped target:

$$\bar{z} = \text{clip}(z^\star, -1, 1). \tag{29}$$

The NSWM-TR objective is

$$\mathcal{J}_z = \|\hat{y}^{\text{TR}} - y\|_2^2 + \lambda_4 \rho\left(z_\theta(\phi), \bar{z}\right) + \lambda_5 \|e_\theta(\phi)\|_1, \tag{30}$$

where ρ is a smooth L_1 loss. The first term fits the next state. The second term encourages the network to express corrections in normalized symbolic coordinates. The third term discourages unnecessary expansion beyond the symbolic trust region.

This formulation yields a clear decomposition of model behavior. If the target lies inside the symbolic region, the model can fit it with $e_\theta \approx 0$ and $z_\theta \approx z^\star$. If the target lies outside the symbolic region, the clipped coordinate loss keeps z_θ near the boundary, while the prediction loss can be reduced by increasing e_θ. Thus, escape behaves as a learned diagnostic of symbolic under-coverage.

Compared with NSWM-GR, NSWM-TR is more conservative. NSWM-GR asks whether a residual improves prediction; NSWM-TR asks whether the correction can be explained as a bounded displacement inside symbolic uncertainty. This makes NSWM-TR especially suitable when the symbolic model provides meaningful uncertainty estimates. It also makes failures easier to interpret: high escape values indicate dimensions or regions where the evolved symbolic laws are insufficient.

H. Comparison Between GR and TR

Both NSWM-GR and NSWM-TR share the same symbolic law library and differ only in how they parameterize Δ_θ. NSWM-GR uses

$$\Delta_\theta(\phi) = g_\theta(\phi) \odot r_\theta(\phi), \tag{31}$$

whereas NSWM-TR uses

$$\Delta_\theta(\phi) = (h + e_\theta(\phi)) \odot z_\theta(\phi). \tag{32}$$

The two corrections encode different assumptions. NSWM-GR assumes that residual errors can be learned directly and that a gate can decide when neural correction is beneficial. NSWM-TR assumes that symbolic uncertainty defines a meaningful local coordinate system around the symbolic prediction.

The GR correction is preferable when symbolic laws capture only a rough trend and residual errors may be large or poorly calibrated. The TR correction is preferable when symbolic laws provide reliable uncertainty and we want the neural model to remain close to interpretable symbolic predictions. In practice, these variants provide complementary trade-offs between flexibility and conservatism.

I. Examples of Generated Laws

We include representative symbolic laws generated by LEAN to illustrate the structure of the learned program library. These examples are taken from NSWM-TR checkpoints after weight fitting, using laws whose fitted weights exceed 0.05. The generated programs are compact, executable Python classes with a precondition and an effect. The precondition restricts the state-action region where the law applies, while the effect writes probabilistic predictions for selected next-state dimensions. These laws are not manually written templates; they are produced by the LLM-guided law evolution procedure and then retained only if they pass validation and receive nontrivial fitted weight.

The first example is a Swimmer law with fitted weight 0.446. It captures three reusable patterns: position integration from velocity, damping of root/body velocities, and action-driven rotor velocities. The law also uses wider prediction radii for root coordinates, reflecting higher uncertainty in global motion.

```
class SynthesizedSwimmerDriveIntegrationTuned:

    def __init__(self, extractor):
        self.extractor = extractor

    def precondition(self, current_state, action):
        return (
            len(current_state.qpos) == 5
            and len(current_state.qvel) == 5
            and len(action.raw) == 2
            and abs(current_state.qvel[0]) > 0.01
        )

    def effect(self, current_state, action):
        dt = current_state.dt
        raw = action.clipped()

        for i in range(5):
            pos = float(current_state.qpos[i]) + float(current_state.qvel[i]) * dt
            radius = 2 if i < 2 else 1
            current_state.predictions[qpos_id(i)] = (
                self.extractor.qpos_distribution(i, pos, radius=radius)
            )

        damping_root = 0.85
        damping_body = 0.7
        damping_rotor = 0.88
        coupling_coeff = 0.09
        drive_gain = 0.7

        rotors_mean = 0.5 * (current_state.qvel[3] + current_state.qvel[4])
        new_qvel_0 = damping_root * current_state.qvel[0] + coupling_coeff * rotors_mean
        new_qvel_1 = damping_root * current_state.qvel[1] + coupling_coeff * rotors_mean
        new_qvel_2 = damping_body * current_state.qvel[2]
```

```
        current_state.predictions[qvel_id(0)] = (
            self.extractor.qvel_distribution(0, new_qvel_0, radius=1)
        )
        current_state.predictions[qvel_id(1)] = (
            self.extractor.qvel_distribution(1, new_qvel_1, radius=1)
        )
        current_state.predictions[qvel_id(2)] = (
            self.extractor.qvel_distribution(2, new_qvel_2, radius=1)
        )

        new_qvel_3 = damping_rotor * current_state.qvel[3] + drive_gain * float(raw[0])
        new_qvel_4 = damping_rotor * current_state.qvel[4] + drive_gain * float(raw[1])

        current_state.predictions[qvel_id(3)] = (
            self.extractor.qvel_distribution(3, new_qvel_3, radius=1)
        )
        current_state.predictions[qvel_id(4)] = (
            self.extractor.qvel_distribution(4, new_qvel_4, radius=1)
        )
```

The second example is a Pusher law with fitted weight 0.946. It separates the robot arm and object-related dimensions. The arm velocities are modeled with damping, action input, and neighboring-joint coupling, while the object and goal coordinates are kept conservative unless stronger evidence is available. This is a useful pattern in object-interaction tasks where most transitions are dominated by arm motion and only some transitions involve contact.

```
class SynthesizedPusherArmVelocityCrossoverLaw:

    def __init__(self, extractor):
        self.extractor = extractor

    def precondition(self, current_state, action):
        return (
            len(current_state.qpos) >= 11
            and len(current_state.qvel) >= 11
            and len(action.raw) >= 7
        )

    def effect(self, current_state, action):
        raw = action.clipped()
        dt = current_state.dt

        damping = 0.9
        coupling_weight = 0.05

        for i in range(7):
            vel = damping * float(current_state.qvel[i])
            vel += (1.0 - damping) * float(raw[i])

            if i > 0:
                vel += coupling_weight * float(current_state.qvel[i - 1])
            if i < 6:
                vel += coupling_weight * float(current_state.qvel[i + 1])

            current_state.predictions[qvel_id(i)] = (
                self.extractor.qvel_distribution(i, vel, radius=2)
            )

        for i in range(7):
            pos_pred = float(current_state.qpos[i]) + float(current_state.qvel[i]) * dt
            current_state.predictions[qpos_id(i)] = (
                self.extractor.qpos_distribution(i, pos_pred, radius=2)
            )

        for i in range(7, 11):
            current_state.predictions[qpos_id(i)] = (
                self.extractor.qpos_distribution(
                    i, float(current_state.qpos[i]), radius=1
                )
            )
            current_state.predictions[qvel_id(i)] = (
                self.extractor.qvel_distribution(
                    i, float(current_state.qvel[i]), radius=1
```

```
            )
        )
```

The final example is an Ant law with fitted weight 0.077. Although each individual Ant law has a smaller fitted weight, the retained library contains many related variants, indicating that the product-of-experts model combines several weak but reusable dynamics programs. This example predicts root velocity from aggregate leg actions, drives leg velocities from matching actions, integrates root and joint positions, and updates the torso quaternion using angular velocity.

```
class SynthesizedAntRootLegQuaternionDynamicsV3:

    def __init__(self, extractor):
        self.extractor = extractor

    def precondition(self, current_state, action):
        return (
            len(current_state.qpos) == 15
            and len(current_state.qvel) == 14
            and len(action.raw) >= 8
            and 0.35 < current_state.qpos[2] < 1.0
        )

    def effect(self, current_state, action):
        dt = current_state.dt
        qpos = current_state.qpos
        qvel = current_state.qvel
        raw = action.clipped()

        damping_root_lin = 0.88
        damping_root_ang = 0.68
        damping_leg_vel = 0.78

        left_leg_action_sum = raw[0] + raw[1] + raw[2] + raw[3]
        right_leg_action_sum = raw[4] + raw[5] + raw[6] + raw[7]
        front_leg_action_sum = raw[0] + raw[1] + raw[4] + raw[5]
        back_leg_action_sum = raw[2] + raw[3] + raw[6] + raw[7]
        avg_leg_vel = sum(qvel[6:14]) / 8.0

        for i in range(3):
            base = damping_root_lin * qvel[i]
            if i == 2:
                pred = base + 0.03 * avg_leg_vel
            elif i == 0:
                pred = base + 0.12 * (left_leg_action_sum - right_leg_action_sum)
                pred += 0.09 * avg_leg_vel
            else:
                pred = base + 0.12 * (front_leg_action_sum - back_leg_action_sum)
                pred += 0.09 * avg_leg_vel

            current_state.predictions[qvel_id(i)] = (
                self.extractor.qvel_distribution(i, pred, radius=2)
            )

        root_ang_pred = [
            damping_root_ang * qvel[3]
            + 0.14 * (front_leg_action_sum - back_leg_action_sum),
            damping_root_ang * qvel[4]
            + 0.14 * (left_leg_action_sum - right_leg_action_sum),
            damping_root_ang * qvel[5]
            + 0.07 * (left_leg_action_sum - right_leg_action_sum),
        ]

        for i in range(3, 6):
            current_state.predictions[qvel_id(i)] = (
                self.extractor.qvel_distribution(i, root_ang_pred[i - 3], radius=2)
            )

        for i in range(8):
            idx = 6 + i
            neighbors = []
            if i > 0:
                neighbors.append(qvel[idx - 1])
            if i < 7:
                neighbors.append(qvel[idx + 1])
```

```
        neighbor_mean = sum(neighbors) / len(neighbors) if neighbors else 0.0
        pred = damping_leg_vel * qvel[idx] + 0.9 * raw[i] + 0.1 * neighbor_mean

        current_state.predictions[qvel_id(idx)] = (
            self.extractor.qvel_distribution(idx, pred, radius=2)
        )

    for i in range(3):
        pred_pos = float(qpos[i]) + float(qvel[i]) * dt
        current_state.predictions[qpos_id(i)] = (
            self.extractor.qpos_distribution(i, pred_pos, radius=1)
        )

    q = np.array(qpos[3:7], dtype=np.float64)
    wx, wy, wz = root_ang_pred
    dt2 = 0.5 * dt

    omega_q = np.array([
        -(wx * q[1] + wy * q[2] + wz * q[3]),
        wx * q[0] + wz * q[2] - wy * q[3],
        wy * q[0] - wz * q[1] + wx * q[3],
        wz * q[0] + wy * q[1] - wx * q[2],
    ])

    q_pred = q + dt2 * omega_q
    norm = np.linalg.norm(q_pred)
    if norm > 1e-8:
        q_pred /= norm
    else:
        q_pred = np.array([1.0, 0.0, 0.0, 0.0])

    for i in range(4):
        current_state.predictions[qpos_id(3 + i)] = (
            self.extractor.qpos_distribution(3 + i, float(q_pred[i]), radius=1)
        )

    for i in range(7, 15):
        idx_vel = i - 1
        predicted = float(qpos[i]) + float(qvel[idx_vel]) * dt
        current_state.predictions[qpos_id(i)] = (
            self.extractor.qpos_distribution(i, predicted, radius=1)
        )
```

These examples show that the generated symbolic laws recover recognizable continuous-control structure: Euler-style position integration, damped velocity dynamics, action-to-joint mappings, local coupling between neighboring joints, conservative object-state priors, and quaternion updates. At the same time, the laws remain approximate and local. This motivates the neural correction stage: the symbolic programs provide interpretable dynamics priors, while the correction networks handle residual continuous effects that are difficult to express with compact laws alone.