Universal Matrix RL with Requisite Variety: Architecture and Behaviour

Artur Kraskov, Monada Dominion

October 2025

Abstract

This paper is the public-facing specification of the `um_ai_with_rand_and_input` reinforcement-learning stack. Rather than releasing source code, we document how the system reproduces the twelve-step Universal Matrix construction while instrumenting variety inflow and attenuation. The environment exposes input-prelude perturbations, streaming disturbances, and per-episode variety metrics that align the build with Ashby's law of requisite variety. We describe the system architecture, geometric validators, curriculum, evaluation harnesses, and empirical behaviour so that the complete method can be reviewed and re-implemented without accessing proprietary assets.

1 Introduction

The Universal Matrix protocol specifies a deterministic twelve-step construction culminating in a self-reflective hexagonal pattern. Our reinforcement-learning (RL) agent executes that protocol inside `um_ai_with_rand_and_input`, a Gymnasium environment whose transition dynamics, validators, and reward shaping mirror the Occam minimality principles and Ashby's cybernetic framing. This paper makes the full design public without distributing code by detailing the environment, the disturbance interfaces, and the monitoring stack that quantify how much variety the regulator absorbs each episode. The goal is to provide a standalone description suitable for peer assessment and faithful re-implementation.

2 Component Overview

The stack is organised as a set of cooperating modules. Table 1 summarises key responsibilities and their locations.

Component	Responsibility
`um_env.UMEnv`	Gymnasium-compatible environment orchestrating state, rewards, and rendering
`env/steps/*.py`	Deterministic executors for the twelve Universal Matrix steps
`env/mixins/validation.py`	Geometric validators (convexity, cycles, concurrency)
`env/mixins/cost.py`	Least-action cost accounting and milestone targets
`toolkit/primitives.py`	Geometry helpers, intersection solvers, plotting utilities
`variety_tools.py`	Input-prelude injection, streaming disturbances, variety metrics
`train.py`	Fast PPO harness with CLI flags for timesteps, passes, and device
`curriculum_trainer.py`	Stage-by-stage PPO trainer passing intermediate states forward
`test_runner.py`	Deterministic evaluation harness with optional retraining and logging
`repair_smoke.py`	Dependency-light repair challenge showcasing variety instrumentation

Table 1: Core modules governing the `um_ai_with_rand_and_input` reinforcement learner.

3 Environment and Variety Instrumentation

3.1 Action Grammar

The action space is `MultiDiscrete([13, 30, 30, 30, 30])`. Slot 0 selects one of 13 primitives (steps 0–12); slots 1–4 provide operand indices. When `UM_STRICT_GATING=1`, the environment coerces the primitive index toward the minimally admissible set for the current curriculum stage, applying a small penalty to reinforce grammar compliance. Operand selection autonomy is controlled via `UM_AGENT_CHOICES`. All step handlers live in `env/steps` and return $(reward, cost, terminated, info)$ tuples.

3.2 Observations

Observations concatenate point coordinates (up to 30 lexicographically ordered points) with structural features: counts of points/edges, a convex quadrilateral indicator, and four local angles. Optional coordinate noise, governed by `UM_OBS_NOISE`, supports robustness experiments.

3.3 Variety Control

`variety_tools.py` introduces two disturbance mechanisms:

- **Input prelude**: when `UM_INPUT_VARIETY=1`, the reset routine injects up to `UM_NOISE_POINTS` points and `UM_NOISE_LINES` segments before the episode begins.
- **Streaming disturbances**: when `UM_STREAM_EVERY`=K, the environment calls `streaming_noise_step` every K steps during an episode, adding an admissible point or line.

The helper class `VarietyMetrics` accumulates attenuation proxies such as the summed log-ratio between the full primitive alphabet and the allowed set, the mean allowed-set size, and the number of coerced actions. Both `test_runner.py` and `repair_smoke.py` record these metrics alongside disturbance counts.

4 Geometric Steps and Validation

Each Occam primitive is implemented by a dedicated handler. Recent updates ensure the rhombus stage produces a pure quadrilateral without diagonals; diagonals are later introduced by `FindIntersectionStep`. Validators in `env/mixins/validation.py` confirm structural properties:

- Triangle detection via clique search.
- Convex quadrilateral checks using oriented area tests.
- Hexagon validation through cycle detection in the `networkx` graph.
- Concurrency and on-segment tests with tolerant analytic geometry.

Least-action costs are computed in `env/mixins/cost.py`. Passing a milestone at minimal cost grants a +15 bonus; exceeding the budget yields −10.

5 Training Pipeline

5.1 Curriculum

`curriculum_trainer.py` executes the progression `triangle` → `quadrilateral` → `diagonal_intersection` → `hexagon`. Each stage trains for a configurable number of PPO timesteps (default 50,000), saves `um_ai_models/ppo_um_model.zip`, and serialises the environment state for the next stage. Disturbance toggles can be activated during training by exporting the relevant environment variables.

5.2 Fast Harness

`train.py` offers a single-stage harness with `-timesteps`, `-passes`, and `-device` flags. The script validates the environment with `check_env`, schedules checkpoint saves, and uses the persistent `memory.py` store to skip known failing action–observation combinations.

5.3 Integration with Variety

To align with Ashby's law, training can be executed with both prelude and streaming disturbances enabled:

```
UM_INPUT_VARIETY=1 UM_NOISE_POINTS=2 UM_NOISE_LINES=1 \
UM_STREAM_EVERY=3 python3 -m um_ai_with_rand_and_input.train --mode train --
timesteps 3000
```

This configuration exposes the policy to ongoing perturbations while the environment logs variety metrics for post hoc analysis.

6 Evaluation and Tooling

6.1 Smoke Tests

extttsmoke_hexagon.py deterministically executes steps 0–12 without PPO dependencies, saving milestone renders in `correct_renders/`. `repair_smoke.py` demonstrates the repair challenge: it injects variety, runs a heuristic policy, and prints per-episode variety metrics. Figure 1 visualises the full twelve-stage progression captured by the smoke test.

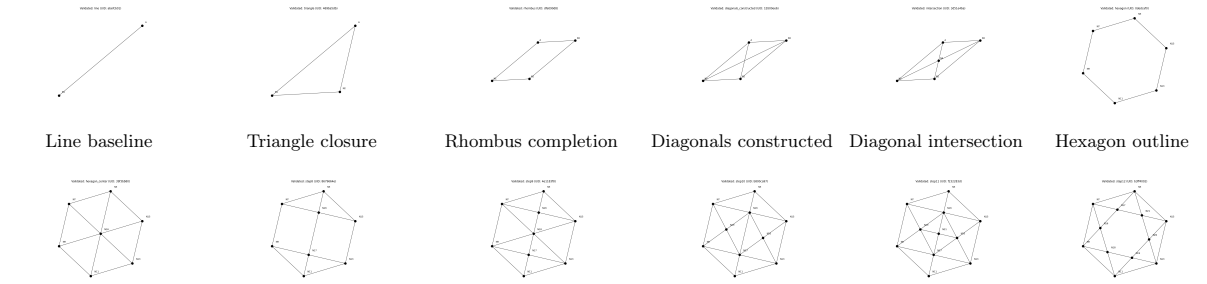

Figure 1: Full smoke-test renders produced by running `python3 -m um_ai_with_rand_and_input.smoke_hexagon`. The twelve milestone images show the deterministic RL-controlled construction from the initial segment through the Star-of-David interior points.

6.2 Test Runner

exttttest_runner.py loads the latest PPO model and runs evaluation episodes, applying a least-ambiguous action sequence to traverse all milestones while logging validation outcomes and variety metrics to `episode_logs.txt`. Streaming disturbances and noise toggles can be applied during evaluation to study regulator behaviour under load.

6.3 Artifacts

Renders are generated via Matplotlib helpers and annotated with UUIDs. Each milestone save is accompanied by a JSON file containing points, edges, milestones, and line definitions, enabling independent verification.

7 Empirical Behaviour

Empirically, policies trained with disturbance toggles maintain milestone progression despite injected variety. `VarietyMetrics` reports attenuation levels that increase with disturbance frequency, matching Ashby's assertion that regulator variety must exceed disturbance variety. When streaming variety is disabled, attenuation drops and coerced actions approach zero, indicating a lower regulatory burden.

8 Alignment with Requisite Variety

The environment exposes both the variety influx (noise points, noise lines, streaming injections) and the regulatory response (allowed-set size, coerced actions). By comparing these signals episode-by-episode, operators can verify that the policy's control complexity adapts to maintain milestone stability. This instrumentation operationalises the cybernetic loop analysed in Kraskov[3], grounding the Universal Matrix construction in a measurable variety budget.

9 Conclusion

The `um_ai_with_rand_and_input` stack couples deterministic geometric rules with explicit variety management. Input preludes and streaming disturbances stress the regulator, while variety metrics quantify the policy's attenuation capacity. Together, these features enable faithful reproduction of the Universal Matrix construction and provide a platform for exploring the law of requisite variety within a symbolic geometry setting.

Acknowledgements

Portions of the design, implementation, and documentation of the extttum_ai_with_rand_-and_input system and this paper were developed with the assistance of AI coding and writing tools, including GitHub Copilot Chat, GPT-5, GPT-5 Codex, and GPT-5.1.

References

[1] A. Kraskov, "Universal Matrix: A Least-Action Cartesian Construction Protocol Embodying Occam's Razor and the Principle of Least Action in Geometric Construction," ResearchGate, 2025. doi:10.13140/RG.2.2.31033.68967. Available at https://www.researchgate.net/publication/394401785_Universal_Matrix_A_Least-Action_Cartesian_Construction_Protocol_Embodying_Occam's_Razor_and_the_Principle_of_Least_Action_in_Geometric_Construction.

[2] W. R. Ashby, *An Introduction to Cybernetics*. Chapman & Hall, London, 1956.

[3] A. Kraskov, "The Universal Matrix as a Cybernetic Design Pattern: Aligning with Ashby's Law of Requisite Variety via Geometric Construction," ResearchGate, 2025. doi:10.13140/RG.2.2.28192.70403.