Symbolic Controller for Non-Linear Dynamic Systems

Project Report

Anir ACHIBANE, Youssef BENTALEB,
Mouad BENSAFIR, Soufiane AIT LHADJ

College of Computing
Mohammed VI Polytechnic University (UM6P)

2025

Contents

1 Introduction

This report details the implementation of a symbolic controller for non-linear dynamic systems. The project aims to synthesize correct-by-construction controllers that guarantee safety and reachability for autonomous systems. Building upon established research in formal methods, we extend the standard pipeline by integrating Large Language Models (LLMs) for natural language problem customization and PyBullet for 3D visualization.

2 Problem Statement

We consider a non-linear dynamical system evolving in discrete time, described by the state-space equation:

$$x(t+1) = f(x(t), u(t), w(t)) \tag{1}$$

where:

- $x(t) \in \mathcal{X} \subset \mathbb{R}^n$ represents the continuous system state at time t.
- $u(t) \in \mathcal{U} \subset \mathbb{R}^m$ represents the control input.
- $w(t) \in \mathcal{W} \subset \mathbb{R}^p$ represents bounded disturbances or uncertainties.

The control objective is to synthesize a feedback controller such that the closed-loop trajectories of the system satisfy a specific high-level specification (e.g., reach a target region while avoiding obstacles) despite the presence of disturbances $w(t)$. Direct synthesis on the continuous model is computationally intractable due to the non-linearities and infinite state space. Therefore, we adopt a **Correct-by-Construction** approach based on symbolic abstraction.

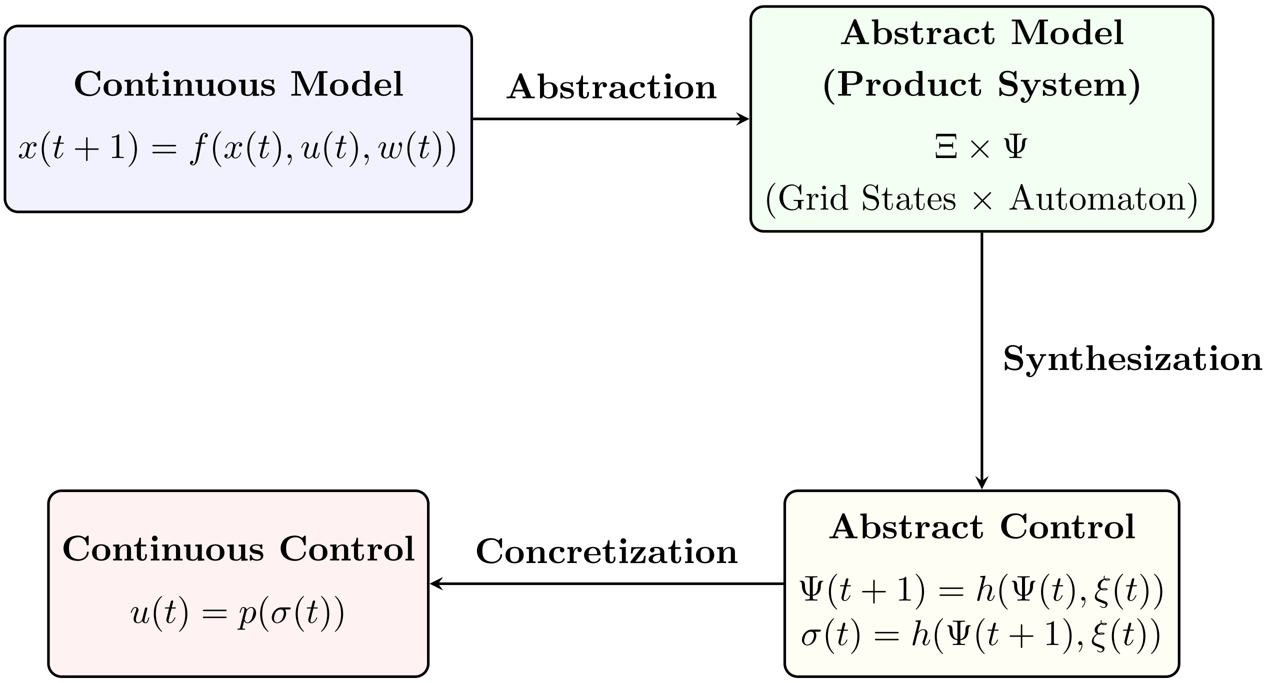

Continuous Model
$x(t+1) = f(x(t), u(t), w(t))$
Abstraction
Abstract Model (Product System)
$\Xi \times \Psi$
(Grid States × Automaton)
Synthesization
Abstract Control
$\Psi(t+1) = h(\Psi(t), \xi(t))$
$\sigma(t) = h(\Psi(t+1), \xi(t))$
Concretization
Continuous Control
$u(t) = p(\sigma(t))$

Figure 1: The Symbolic Control Pipeline: From Continuous Dynamics to Discrete Synthesis.

1. **Abstraction:** The continuous state and input spaces $\mathcal{X}$, $\mathcal{U}$ are discretized into finite sets of states (states ξ and inputs σ) with a finite transition system combined with a specification automaton (states Ψ) to form a Product System.
2. **Synthesization:** We treat the control problem on the abstract graph. Using iterative algorithms to ensure reachability and safety, we synthesize a controller for the discrete system h.
3. **Concretization:** The abstract control strategy is mapped back to the continuous domain using a refinement function $p(\cdot)$. Resulting in a continuous input $u(t) = p(\sigma(t))$ applied to the physical robot.

3 Implementation of the 3-Dimensional Model

In the following section, we focus on our implementation of the symbolic controller for a 3-dimensional model. The dynamics are defined by:

$$x_{1,t+1} = x_{1,t} + \tau(u_{1,t}\cos(x_{3,t}) + w_{1,t}) \tag{2}$$

$$x_{2,t+1} = x_{2,t} + \tau(u_{1,t}\sin(x_{3,t}) + w_{2,t}) \tag{3}$$

$$x_{3,t+1} = x_{3,t} + \tau(u_{2,t} + w_{3,t}) \pmod{2\pi} \tag{4}$$

where τ is the sampling time.

The implementation of the 2-dimensional is similar with very few differences that we discuss in its own section.

3.1 Abstraction of Continuous Space

The continuous state space $\mathcal{X} = [0, 10] \times [0, 10] \times [-\pi, \pi]$ was discretized into a finite grid. We utilized a resolution of 100×100 positions and 30 orientation angles, resulting in a total of 300,000 abstract states.

3.1.1 Integer Indexing Optimization

Storing states as tuples (e.g., (x, y, θ)) or Python sets results in high memory overhead and computational cost. To address this, we implemented 1-dimensional Integer Indexing, mapping the 3D grid coordinates (i, j, k) to a single flat integer.

The mapping function is defined as:

$$\text{idx} = i \times \text{STRIDE\_X1} + j \times \text{STRIDE\_X2} + k \tag{5}$$

This optimization significantly reduced RAM usage from 64GB+ to around 8GB and improved lookup speeds during transition computations.

3.1.2 Transition Table Construction

The transitions between abstract states were computed using the **Growth Bound method**. For every state-action pair (ξ, σ), we compute an over-approximation of the reachable set from the continuous cell $\mathbb{X}_\xi$ and map it to the intersecting discrete states in the grid.

The reachable set is approximated using the growth bound function:

$$f(\text{cl}(\mathbb{X}_\xi), u_\sigma, \mathbb{W}) \subseteq [f(x^*, u_\sigma, w^*) - D_x\delta_x - D_w\delta_w, \quad f(x^*, u_\sigma, w^*) + D_x\delta_x + D_w\delta_w] \tag{6}$$

where:

- x^* and w^* are the centers of the state cell and disturbance set, respectively.
- δ_x and δ_w are the semi-widths (radius) of the state cell and disturbance set.
- D_x and D_w are the bounds on the partial derivatives of the system dynamics f with respect to state x and disturbance w.

Due to the high number of transitions (300,000 states $\times$ 15 commands), we parallelized this computation across CPU cores using `joblib`.

3.2 Controller Synthesis

3.2.1 Product System

The synthesis is performed on a **Product System** that combines the abstraction of the physical dynamics with the specification automaton. We define the product state Ψ_t as a tuple containing the symbolic system states ξ_t and the specification automaton (DFA) states q_t:

$$\Psi_t = (\xi_t, q_t) \in \Xi \times Q \tag{7}$$

The dynamics of this product system are derived by synchronizing the symbolic model transitions g with the DFA transitions δ:

$$\Psi_{t+1} \in G(\Psi_t, \sigma_t) \iff \begin{cases} \xi_{t+1} \in g(\xi_t, \sigma_t) \\ q_{t+1} = \delta(q_t, L(\xi_{t+1})) \end{cases} \tag{8}$$

where $L(\xi)$ maps a physical state to its corresponding region label (e.g., "Region A", "Obstacle").

3.2.2 Safety and Reachability Algorithms

To synthesize the controller, we employ iterative algorithms based on the **Predecessor** operator. For a target set of product states S, $Pre(S)$ is defined as the set of states from which the system can be forced into S despite disturbances:

$$Pre(S) = \{\Psi \in \Xi \times Q \mid \exists \sigma \in \Sigma, G(\Psi, \sigma) \subseteq S\} \tag{9}$$

Crucially, the condition $G(\Psi, \sigma) \subseteq S$ requires that *all* possible non-deterministic successors of the action σ lie within the target set S.

We implemented two specific synthesis algorithms using this operator:

1. **Safety Algorithm:** We compute the maximal controlled invariant set R^* by iteratively removing states that can violate safety constraints (e.g., hitting an obstacle or a DFA trap state). The iteration proceeds as follows:

$$R_0 = Q_{safe}, \quad R_{k+1} = R_k \cap Pre(R_k) \tag{10}$$

   The algorithm terminates when a fixed point $R_{k+1} = R_k$ is reached.

2. **Reachability Algorithm:** We compute the backward reachable set W^* by accumulating states that can force the system into the goal states Q_{goal}:

$$W_0 = Q_{goal}, \quad W_{k+1} = W_k \cup Pre(W_k) \tag{11}$$

   This iteration terminates when $W_{k+1} = W_k$.

3.2.3 Implementation Challenges and Optimizations

The strict inclusion condition in the Predecessor operator ($G(\Psi, \sigma) \subseteq S$) presents a significant challenge. With a coarse discretization, the over-approximation of reachable sets is often too large to fit entirely within safe regions, leading to an empty controller. To resolve this, we use the grid resolution $100 \times 100 \times 30$.

However, this high resolution introduced a time complexity bottleneck. We addressed this with two key optimizations:

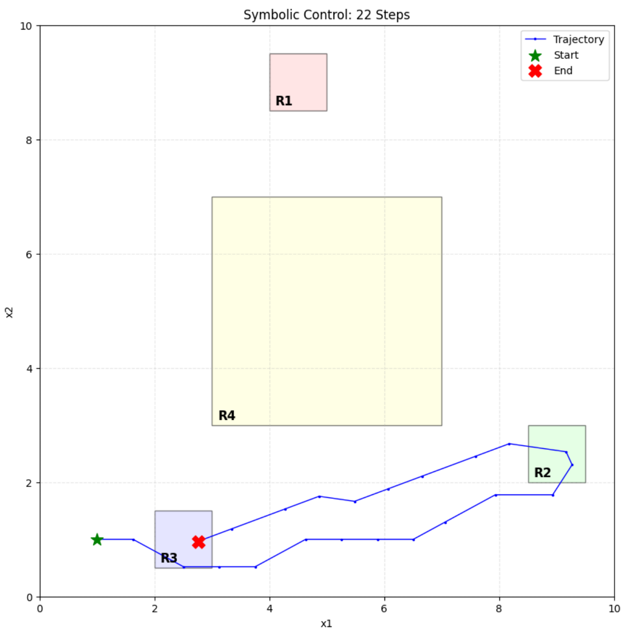
1. **Parallelization:** We utilized multi-core processing (via the `joblib` library) to construct the transition table and perform safety checks in parallel chunks, reducing initialization time.

2. **BitMasking:** To optimize the fixed-point iterations, we replaced standard set operations with **Boolean BitMasks**:

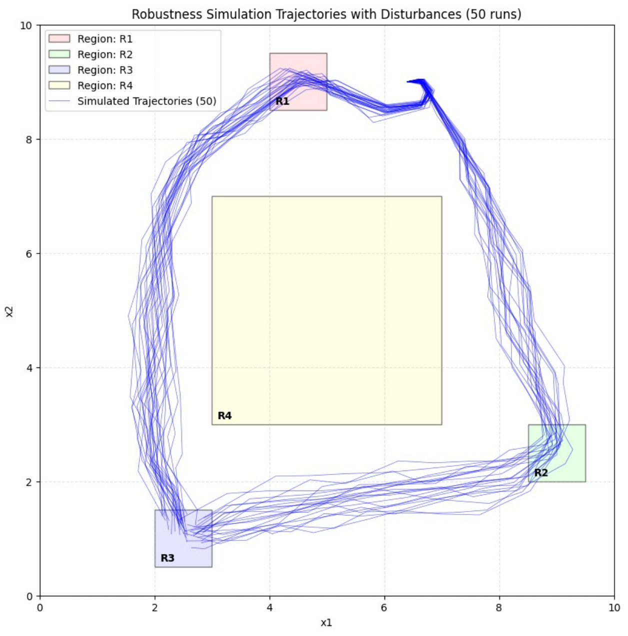
   - Sets of states (e.g., R_k) are represented as binary arrays (or `numpy` boolean arrays) where the i-th bit is true if state i is in the set.
   - Expensive set intersections ($\cap$) and unions ($\cup$) are replaced by fast bitwise AND (&) and OR ($||$)*operations*.
   - Validity checks for successors are performed by direct index lookup on the mask (e.g., `if not valid_mask[next_state_idx]`), eliminating the overhead of hash-set lookups.

4 Visualization

4.1 Static Trajectory



(a) Nominal direct trajectory (no disturbance).



(b) 50 simulations with added random disturbance.

Figure 2: Idealize discrete trajectory and multiple simulated runs under disturbed dynamics

First, we visualized the synthesized controller's output using `matplotlib` on the paper's problem. This allowed us to verify the logical correctness and robustness of the discrete path.

Furthermore, to understand the scope of the synthesized solution, we generated a coverage map (Figure 3). This map visualizes the "winning set" of states—the regions in the state space from which the controller guarantees reaching the goal safely.

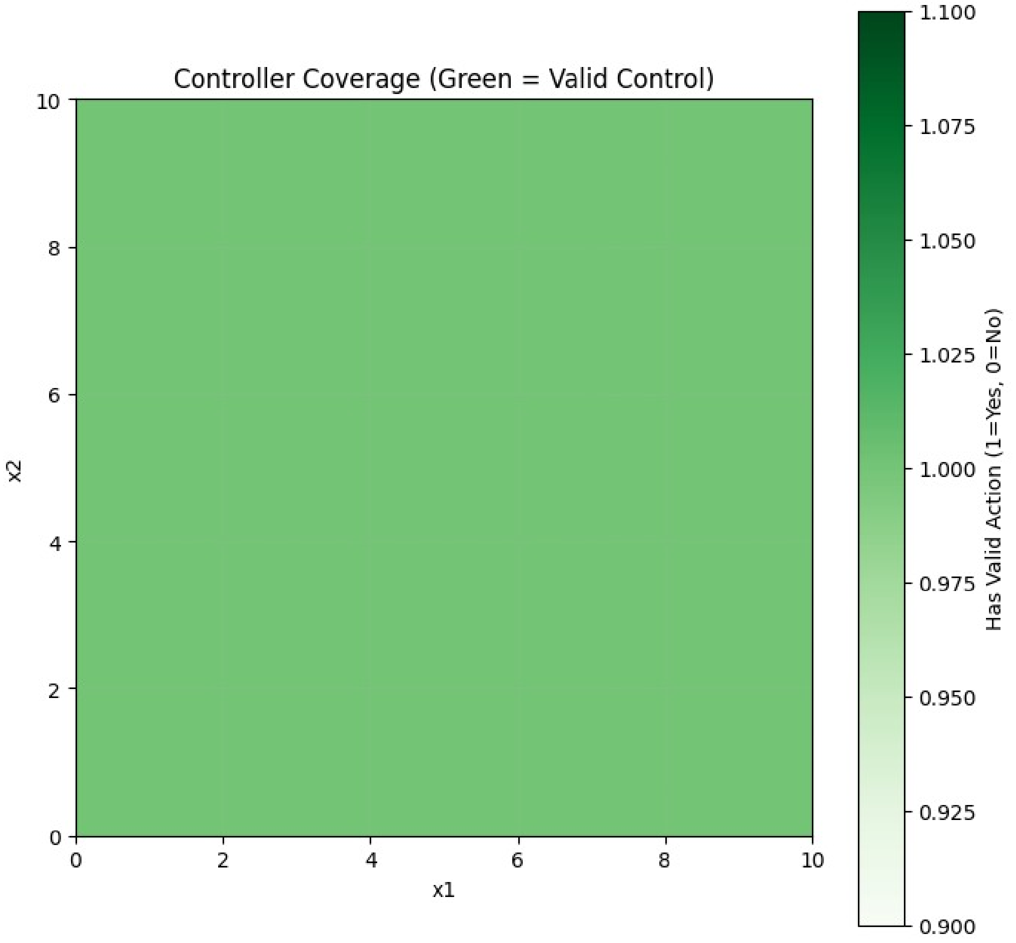


Figure 3: Controller coverage map illustrating the winning set of states from which the goal is guaranteed to be reachable safely.

4.2 PyBullet Simulation

To ensure the controller performs correctly under more realistic constraints, we integrated a physics-based simulation using **PyBullet**.

4.2.1 Simulation Pipeline

The visualization pipeline operates asynchronously from the main synthesis loop to ensure performance:

1. **Trajectory Generation:** The symbolic controller generates a sequence of discrete abstract states $\xi_0, \xi_1, \ldots, \xi_N$.
2. **Concretization & Serialization:** These states are mapped to continuous coordinates (x, y, θ) and serialized into a `sim-traj.pkl` file along with the region definitions.

3. **Playback:** The independent PyBullet script loads the environment (plane, textures, and a Husky robot URDF) and reads the trajectory file.

4.2.2 Smoothing and Interpolation

Since the symbolic controller provides sparse discrete waypoints, directly teleporting the robot between states causes jerky, unnatural motion. To solve this, we implemented a custom interpolation algorithm within the animation loop.

For every pair of consecutive waypoints $P_i = (x_1, y_1, \theta_1)$ and $P_{i+1} = (x_2, y_2, \theta_2)$, we generate intermediate frames using a linear interpolation parameter $\alpha \in [0, 1]$ over a fixed number of steps (e.g., 20 frames per waypoint).

Positional Interpolation: The position is updated linearly at every frame to ensure constant velocity between nodes:

$$x(\alpha) = (1 - \alpha)x_1 + \alpha x_2 \tag{12}$$

$$y(\alpha) = (1 - \alpha)y_1 + \alpha y_2 \tag{13}$$

Heading Interpolation with Blending: To mimic the non-holonomic constraints of a real robot (which often moves before turning or turns while moving), we do not interpolate angular changes immediately. Instead, we define a *blend threshold* $\beta = 0.65$.

- **Phase 1 ($\alpha < \beta$):** The robot maintains its initial heading θ_1 while translating.
- **Phase 2 ($\alpha \geq \beta$):** The robot blends its rotation towards the target heading θ_2 only during the final 35% of the movement segment.

The angular difference is calculated using the shortest path to avoid unnecessary full rotations:

$$\Delta\theta = ((\theta_2 - \theta_1 + \pi) \bmod 2\pi) - \pi \tag{14}$$

The instantaneous heading $\theta(\alpha)$ is updated as:

$$\theta(\alpha) = \begin{cases} \theta_1 & \text{if } \alpha < \beta \\ \theta_1 + \left(\frac{\alpha - \beta}{1 - \beta}\right) \Delta\theta & \text{if } \alpha \geq \beta \end{cases} \tag{15}$$

This technique prevents the "tank turn" visual artifact (rotating in place before moving) and results in smooth, arc-like trajectories as the robot approaches goal regions.

5 LLM Integration

A novel contribution of this project is the integration of Large Language Models (LLMs) to bridge the gap between natural language user intent and formal system specifications. This allows non-expert users to define complex control tasks without needing to manually construct automata or geometric regions.

5.1 Pipeline Architecture

We utilized the `gemini-2.5-pro` model to process user inputs. The integration pipeline follows a structured workflow to ensure reliability:

1. **User Input:** The user provides a high-level command (e.g., "Go to the center, then top right, avoiding region R4").

2. **System Prompting:** This input is wrapped in a comprehensive system prompt that enforces strict output formatting and logical consistency rules.

3. **JSON Generation:** The LLM generates a structured JSON object containing both the geometric definitions and the logical automaton.

4. **Parsing & Synthesis:** The backend parses this JSON to construct the specification automaton (DFA) and map regions to the grid, which are then passed to the synthesis engine.

5.2 Output Structure

The LLM is instructed to return a strictly formatted JSON object. This structure is critical for parsing the automaton programmatically. The key components are:

- **Regions:** A dictionary defining named rectangular zones with their (x, y, θ) bounds.
- **DFA Definition:**
  - `states`: A list of all automaton states (e.g., $q_0, q_1, q_{trap}, q_{accept}$).
  - `alphabet`: The set of region labels that trigger transitions.
  - `transitions`: A list of transition rules in the format {`"from": "q0", "input": "A", "to": "q1"`}.
  - `start_state` & `accepting_states`: Defining the initial and goal configurations.

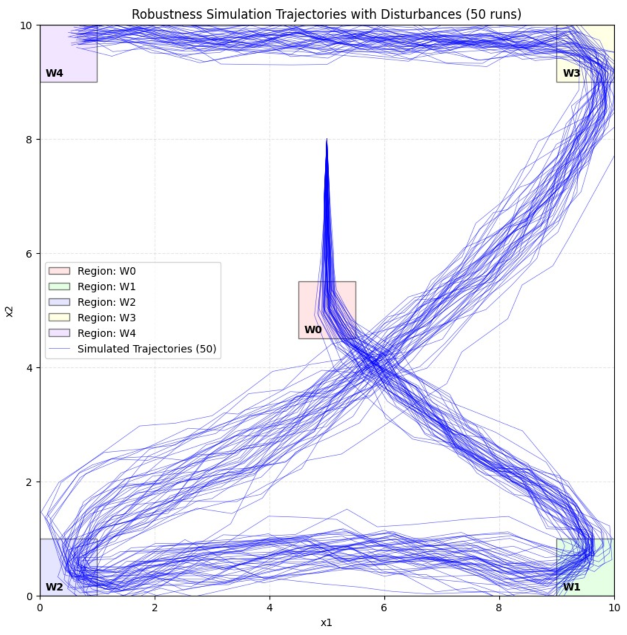
5.3 Robustness and Hallucination Mitigation

A major challenge with LLMs is "hallucination," where the model generates invalid or physically impossible specifications. To mitigate this, we engineered a **System Prompt** that acts as a strict firewall. It includes:

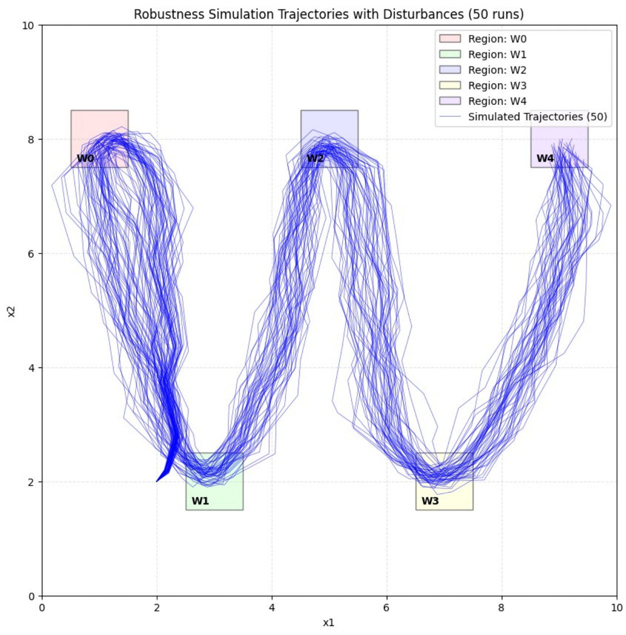
- **Geometric Sanity Checks:** Explicit constraints on the workspace bounds (e.g., $x \in [0, 10]$).
- **Automaton Completeness:** Requiring that every state handles every possible input (including a self-loop for "None" when no region is visited).
- **Chain of Thought:** Instructing the model to output a reasoning trace before generating the JSON, which improves logical accuracy for complex multi-step tasks.

We validated this robustness with three distinct scenarios, ranging from replication of standard benchmarks to handling vague, abstract instructions.

For the first example, we prompted the following: Go to the center of the map, then to the bottom right, then the bottom left, then the top right, then the top left and stop. For the second prompt, we opted for a more vague and ill-described request: Draw W on the map.



(a) Replication of the standard multi-region traversal task.



(b) Processing vague user intent (”Draw a W on the map”) into geometric waypoints.

Figure 4: Idealize discrete trajectory and multiple simulated runs under disturbed dynamics

6 Concluding Remarks

6.1 2-Dimensional Implementation

It is worth noting that for simpler, 2-dimensional versions of this problem (e.g., controlling a robot on a plane without orientation constraints), the computational complexity is significantly lower. In a 2D state space ($\mathcal{X} \subset \mathbb{R}^2$), the number of abstract states scales quadratically rather than cubically. Consequently, the naive ”brute-force” approach to computing transitions and safety sets is sufficient. The advanced optimizations required for the full 3D model—specifically parallelization and bitmasking are not strictly necessary in 2D, as the total number of calculations remains manageable for standard processors.

6.2 Summary

This report presented the end-to-end implementation of a symbolic control framework for non-linear dynamic systems. We began by formalizing the control problem as a game played on a discretized abstraction of the continuous state space. We successfully constructed a **correct-by-construction controller** capable of guaranteeing safety and reachability for a 3-dimensional unicycle model.

Key achievements of this project include:

- **Scalable Abstraction:** We overcame the curse of dimensionality inherent in 3D discretizations ($100 \times 100 \times 30$ grid) by implementing **Integer Indexing** for memory efficiency and **Parallelization** to speed up transition table construction.
- **Optimized Synthesis:** To handle the massive state space during synthesis, we replaced standard set operations with high-performance **BitMasking** techniques, enabling efficient fixed-point iterations for safety and reachability algorithms.

- **Natural Language Interface:** We integrated a **Large Language Model (LLM)** pipeline that translates vague user commands into rigorous formal specifications (regions and automaton), protected by a robust system prompt to prevent hallucinations.
- **Visualization:** We validated our discrete controller using **PyBullet**, a physics-based engine. A custom interpolation scheme was developed to smooth the discrete waypoints, resulting in realistic, continuous robot motion.

In conclusion, this project demonstrates that symbolic control, often viewed as purely theoretical, can be engineered into a practical, interactive, and visually verified toolchain for autonomous systems.