

Final Assignment: Policy Learning

Luca Podavini 257844

Matteo Grisenti 257855

January 2026

Disclaimer: This assignment is a custom one defined with the professor Andrea Del Prete

1 Introduction

The objective of this assignment is to explore Policy Learning techniques for developing a robotic controller. Given the constraints in hardware and time, the project focuses on the control of a standard Dubins mobile robot. The assignment is divided into two primary objectives:

- **Trajectory Tracking Policy:** The first scenario involves training a neural network to control the robot as it follows a predefined path consisting of a set of waypoints. The navigation pipeline consists of a Dubins Planner to define the path, followed by the learned policy to execute the movement. For this scenario, we implement a classical Reinforcement Learning approach, incorporating a variant that utilizes an estimated Value Function baseline to reduce variance.
- **Navigation Policy:** In the second scenario, the trajectory is removed. Instead, a single end-to-end policy is developed to navigate the environment. Given the robot's state and environmental data (including obstacles), the policy must guide the robot to a final goal. To manage this increased complexity, we explore Actor-Critic theory by implementing the Proximal Policy Optimization (PPO) algorithm.

2 Environment Design: UnicycleEnv

The primary step in our development was the implementation of a simulation environment to facilitate training and evaluation. This was built using the **OpenAI Gymnasium** framework, which provides a standardized interface for reinforcement learning agents.

The environment, implemented as `UnicycleEnv` within the `envs/unicycle_env.py` module, consists of a rectangular 2D workspace containing a robot, a goal pose, and a set of obstacles. The environment is characterized by the following components:

- **Observation Space:** Represents the robot's *ego-centric* state $[\rho, \alpha, \delta\theta]$ relative to the goal. This includes:
 - ρ : The Euclidean distance (radial error) between the robot's center and the target coordinates.
 - α : The *look-ahead* angle or bearing error; specifically, the angle between the robot's current heading and the vector pointing toward the goal.
 - $\delta\theta$: The relative heading error; the angular difference between the robot's current orientation and the desired final orientation at the goal.
- **Action Space:** A continuous space representing the robot inputs (v, ω) : linear velocity (v) and angular velocity (ω).

2.1 Modularity and Configuration

To ensure flexibility, the environment is fully modular. Its parameters are managed via YAML configuration files located in the `/configs` directory. This architecture allows the user to toggle between **fixed** scenarios for benchmarking and **randomized** spawns for robust policy generalization.

2.2 Core Functionality

Following the Gymnasium API, we implemented the following key methods:

- `reset(seed)`: Initializes a new episode. The use of a pseudo-random seed ensures the reproducibility of obstacle and goal placements, which is critical for debugging and performance comparison.
- `step(action)`: Executes one simulation step. It takes the agent's velocity commands as input and updates the robot's pose by integrating the non-holonomic unicycle dynamics over a discrete time step Δt . It returns the new observation, the termination flags (collision or goal reached) and a general info object.

Finally, a visualization tool was developed using the **OpenCV** library, providing a real-time graphical representation of the robot's movements, the obstacles, and the trajectory buffer.

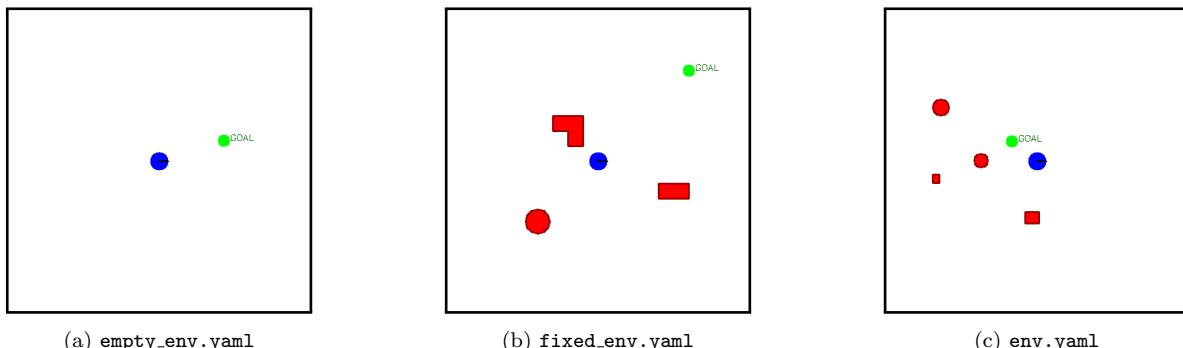


Figure 1: Three renders of the Unicycle Environment.

3 Trajectory Tracking Policy

3.1 Dubins Planner

To facilitate trajectory tracking, we implemented a path planning entity responsible for generating the reference trajectory for the controller. We utilized a standard **Dubins Planner**, which computes the shortest kinematically feasible path (as a set of waypoints) between a start and goal pose (x, y, θ) given a minimum turning radius.

The planner, implemented as the `DubinsPlanner` class in `planner/dubins_planner.py`, evaluates the six classic Dubins primitives (LSL, RSR, LSR, RSL, RLR, LRL) to identify the optimal path. The resulting trajectory is returned as a **ordered set of waypoints**. The density of this set is determined by a granularity parameter (step size), which ensures the path is sufficiently discretized for the controller to follow smoothly.

NB: Since the primary focus of this stage is policy learning for tracking rather than obstacle avoidance, we utilize the `empty_env.yaml` configuration. By removing obstacles, we ensure the planner provides a collision-free reference, allowing us to evaluate the controller's tracking performance in isolation.

3.2 Lyapunov Controller: Baseline

To provide a rigorous benchmark for the learned policies, we implemented a deterministic controller based on **Lyapunov Stability Theory**. This controller serves as a classical control baseline, allowing us to compare the efficiency of the Reinforcement Learning agents against a mathematically grounded approach.

The controller, implemented in `controllers/lyapunov_controller.py`, operates by receiving the **tracking ego-centric observation** as input. Unlike the general environment observation, this specific measure is calculated relative to the **next active waypoint** in the path provided by the planner. By minimizing the error relative to this local target, the controller ensures asymptotic stability along the entire trajectory.

- **Linear Velocity Control (v):** $v = v_d + K_P \cdot \rho \cdot \cos(\alpha)$
 - v_d is the desired velocity (calculated as $0.5 \cdot \tanh(\rho)$ when a reference is absent)
 - K_P is the distance gain
- **Angular Velocity Control (ω):** $\omega = \omega_d - \frac{v_d \cdot \rho}{\cos(\delta\theta/2)} \sin(\alpha + \pi + \frac{\delta\theta}{2}) - K_\theta \cdot \sin(\delta\theta)$
 - ω_d is the desired velocity
 - the first part compensates for lateral displacement relative to the target
 - K_θ is the distance gain

By tuning the gains K_P and K_θ , this controller provides a reliable "ideal" trajectory for the *Trajectory Tracking* scenario.

3.3 Evaluation Methodology

Following the implementation of the Lyapunov controller as a baseline, we developed a comprehensive evaluation suite designed to assess individual controller performance and facilitate a direct, rigorous comparison against the baseline. Under this framework, each controller is evaluated over 50 discrete episodes. Each episode is defined by a unique environmental configuration and a reference path generated via a Dubins Planner, utilizing a randomized minimum turning radius to ensure a diverse set of navigational challenges.

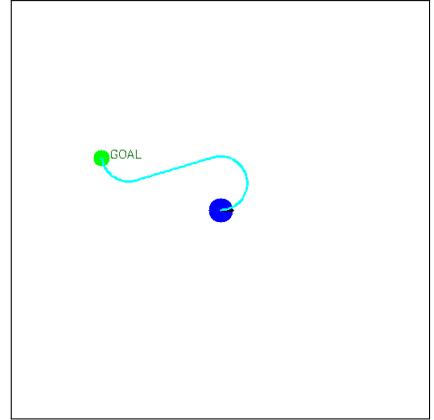


Figure 2: Dubins Path Example

To ensure reproducibility and maintaining a fair benchmark across different control strategies, we employ a fixed set of 50 evaluation seeds. These seeds govern the initialization of both the environmental layout and the planner's kinematic constraints. Crucially, these evaluation seeds are strictly excluded from the training phase to maintain a clear separation between training and testing datasets.

It should be noted, however, that due to the relatively low complexity of the environment, distinct seeds may occasionally generate configurations that are qualitatively similar to those encountered during training. While this implies that total environmental novelty cannot be strictly guaranteed, the use of held-out seeds remains a fundamental practice to evaluate the generalization capabilities of the trained policy.

Performance Metrics

During each simulation episode, we collect a suite of metrics to quantify the efficiency, accuracy, and stability of the controller:

- **Success Rate:** The percentage of episodes where the robot successfully reaches the goal state within a predefined spatial threshold.
- **Cross Track Error (CTE):** A measure of the lateral deviation of the robot from the reference path. We calculate the instantaneous distance between the robot's center and the line segment connecting the two nearest waypoints.
- **Tortuosity:** Defined as the ratio of the total distance traveled by the robot ($L_{traveled}$) to the theoretical length of the planned path (L_{path}):

$$\tau = \frac{L_{traveled}}{L_{path}}$$

Values near 1.0 indicate high path fidelity,

Values significantly greater than 1.0 suggest oscillations or inefficient maneuvering,

Values significantly lower than 1.0 indicate "cutting" across curvatures.

- **Smoothness:** Computed as the mean absolute difference between the angular velocity (ω) commands of consecutive time steps. Lower values indicate more stable and fluid control.
- **Energy Consumption (Proxy):** An approximation of energy expenditure calculated as the sum of squared linear (v) and angular (ω) velocities multiplied by the time step (Δt):

$$E \approx \sum_{t=0}^T (v_t^2 + \omega_t^2) \Delta t$$

- **Average Velocity and Steps:** The mean linear speed and the total number of discrete simulation steps required to complete the path, representing the temporal efficiency of the navigation.

The results are aggregated across the 50 episodes. For each metric, we report the mean and one standard deviation ($\mu \pm \sigma$) to characterize both the expected performance and the robustness.

Lyapunov Controller Evaluation

The following results characterize the performance of the Lyapunov controller, which serves as the established baseline for this study.

Navigation Performance		Path Fidelity		Control Efficiency	
Metric	Value	Metric	Value	Metric	Value
Status	EXCELLENT	Mean CTE	0.0543 ± 0.0089 m	Avg Velocity	0.09 ± 0.02 m/s
Success Rate	100.0 %	Max CTE (Avg)	0.0823 m	Smoothness	0.1158 ± 0.0014
Steps (Avg)	414.7 ± 59.2	Tortuosity	0.971 ± 0.005	Avg Energy	1.18 ± 0.19

Table 1: Lyapunov Controll Evaluation

3.4 Trajectory Tracking Network

For the *Trajectory Tracking* task, we implemented within `models/trajectory_tracking_network.py` a Multi-Layer Perceptron (MLP), designed to map ego-centric observations to a continuous action distribution.

The so called TTNetwork architecture is defined by the following structural choices:

- **Input Layer:** The network accepts a 3-dimensional ego-centric state vector as input. The first linear layer transforms this input into a 128-dimensional hidden feature space.
- **Hidden Layer:** The model employs a hidden layer of 64 neurons, mapping the 128-dimensional hidden vector into a 64-dimensional space to extract higher-level features.
- **Output Layer:** The final layer maps the 64-dimensional hidden features into a 2-dimensional output vector, representing the linear and angular velocity pair (v, ω) .
- **Activation Function:** The **Tanh** activation function is used between all layers to ensure smooth gradients. Additionally, a Tanh activation is applied to the output layer to ensure the control commands are naturally bounded within the $[-1, 1]$ range, matching the environment's requirements.
- **Action Mean (μ):** The output of the MLP is interpreted as the *action mean*, representing the most probable action the policy will take given a specific input state.
- **Learnable Exploration (σ):** To facilitate stochastic exploration, we implemented a learnable `log_std` parameter. Consequently, the network defines a Gaussian distribution $\mathcal{N}(\mu, \sigma)$, allowing the agent to internally tune its exploration pressure throughout the training process.

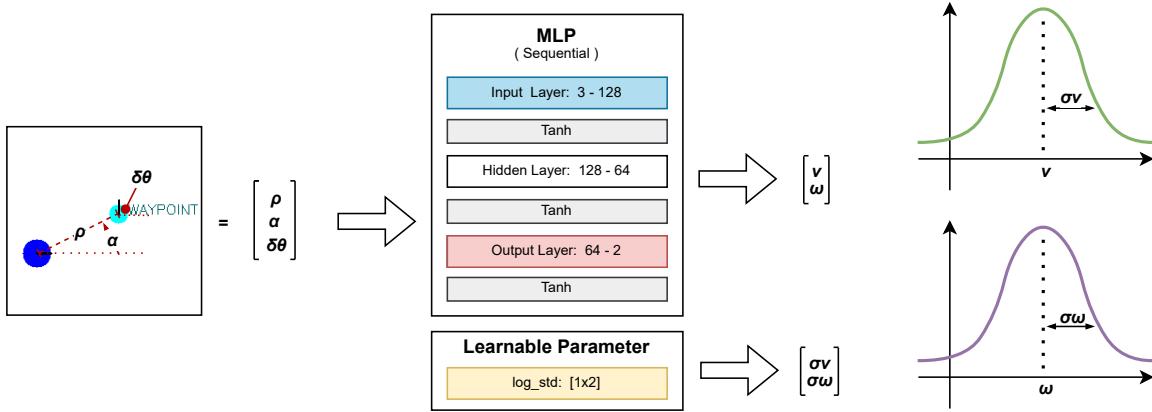


Figure 3: TTNetwork Architecture

3.5 REINFORCE Algorithm

The TTNetwork is trained using the **REINFORCE** algorithm, a Monte Carlo Policy Gradient method. Our implementation follows a structured sequence of steps for each training iteration:

1. **Episode Setup:** At the start of each episode, a new random seed is sampled to ensure environmental variety. A smooth reference trajectory is then generated using the **Dubins Planner** with randomized curvatures to improve the policy's ability to generalize across different path shapes.
2. **Interaction Loop:** Throughout the episode, the following steps are performed at each discrete time step:
 - *State Transformation:* While the default environment provides an observation relative to the final goal, a tracking policy must learn to follow a path by reaching sequential waypoints. Consequently, we map the raw environment observation into an **Ego-Centric Observation** relative to the next path waypoint.

- *Action Selection*: The policy network processes the state and returns the parameters of a Gaussian distribution. An action pair (v, ω) is then sampled from this distribution.
- *Environment Step*: The robot executes the sampled action, updates its pose according to the unicycle kinematics, and receives the updated environment state.
- *Reward Computation*: A scalar reward is calculated based on tracking precision, motion smoothness, and progress along the path.
- *Terminal Conditions*: An episode terminates if the agent successfully reaches the final goal, triggers a collision, exceeds the maximum lateral error threshold (error > 2.0), or reaches the maximum step limit.

3. **Batch Learning**: Upon collecting a specified `BATCH_SIZE` of episodes, the **Update Step** is performed. This step computes the policy loss and performs backpropagation to update the **Policy Network** parameters.

Additionally, we implemented an **Early Stopping** logic, which halts training if the average batch reward fails to improve over a predefined `PATIENCE` period. Parallel to this, a **Checkpointing** mechanism ensures that we store the "best" version of the network weights (those achieving the highest average batch reward) rather than simply the final iteration.

The training pipeline described above is implemented in the notebook `training/reinforcement.ipynb`. Below, we detail the specific components of this pipeline.

3.5.1 Reward Function Design

To guide the agent toward efficient and smooth trajectory tracking, we designed a composite reward function R_{total} , consisting of continuous *dense* rewards for constant feedback and discrete *event* rewards for high-level task completion.

Dense Tracking Rewards: These encourage the robot to minimize errors relative to the reference path:

- **Position Reward** (r_{pos}): We use a Gaussian kernel to provide a smooth gradient that peaks when the distance error ρ is zero:

$$r_{pos} = \exp(-k_\rho \cdot \rho^2)$$

- **Heading Reward** (r_{head}): Encourages alignment with the target waypoint direction using a cosine similarity metric:

$$r_{head} = \cos(\delta\theta)$$

- **Smoothness Penalty** (r_{smooth}): Penalizes high angular velocities to discourage jittery steering:

$$r_{smooth} = -|\omega|$$

- **Velocity Reward** (r_{vel}): Incentivizes forward progress, but only if the heading error is small ($|\delta\theta| < 1.0$), and penalizes stasis ($v < 0.1$) when aligned.

Discrete Event Rewards: These signify major milestones or failures:

- **Checkpoint Progress**: A reward $R_{CHECKPOINT} = 10.0$ is granted for every path index advanced.
- **Success**: A large sparse reward $R_{GOAL} = 200.0$ is awarded upon reaching the final destination.
- **Failure Penalties**: Large negative penalties ($R = -200.0$) are applied for collisions or wandering off-path ($\rho > 2.0$).

The final step reward is the weighted sum:

$$R_{step} = (w_\rho r_{pos} + w_\theta r_{head} + w_\omega r_{smooth} + w_v r_{vel}) + R_{events}$$

Implementation Note: The reward logic is modularized in the utility file `utils/reward.py`.

3.5.2 Policy Update Mechanism

The `update()` function implements the stochastic gradient ascent step for the REINFORCE algorithm, transforming trajectory data into policy improvements:

- **Exploiting Causality:** From the collected rewards, we derive the **Return-to-Go** (G_t) by exploiting the principle of causality (actions only affect future rewards):

$$G_t = \sum_{k=t}^T \gamma^{k-t} R_k$$

- **Batch Normalization:** Before calculating the loss, the returns G_t are normalized across the batch to stabilize updates and prevent vanishing or exploding gradients.
- **Policy Loss Calculation:** Following the policy gradient theorem, the loss is computed as the negative log-likelihood of the actions weighted by the normalized returns:

$$L_{policy} = -\frac{1}{N} \sum_{t=1}^N \left(\log \pi_\theta(a_t | s_t) \cdot \hat{G}_t \right)$$

- **Gradient Regularization:** We apply **Gradient Clipping** with a maximum norm of 1.0 to ensure robust convergence and prevent destabilizing parameter shifts.
- **Optimization:** We utilize the **Adam** optimizer for backpropagation. After the update, the batch buffer is cleared to begin a new cycle of environment interaction. With the optimizer we add also a schedule that reduce the learning rate while the training go forward.

3.5.3 Training Configuration and Results

Following an extensive hyperparameter tuning phase, a final configuration was established to optimize the convergence of the policy. The training parameters and reward weights are summarized in the tables below:

Training Hyperparameters				Reward Hyperparameters			
MAX_EPISODES	4000	BATCH_SIZE	32				
MAX_STEPS	200	GAMMA (γ)	0.99	w_ρ	1.0	R_{GOAL}	200.0
POLICY_LR	1×10^{-3}	PATIENCE	10	w_θ	0.8	$R_{COLL.}$	-200.0
				w_ω	0.1	$R_{OFF_P.}$	-200.0
				w_v	2.0	$R_{CHECK.}$	10.0

The training session, including model checkpoints and logs, is stored in the directory `training/v1_no_baseline`. The results of this training phase are illustrated in Figure 6 and in the evaluation Table 2

Training Observations

- **Training Variance:** As illustrated in the Reward Progress plot (Figure 6), the training process exhibits significant variance. This instability manifests as consecutive "bad" batches followed by a sudden "good" batch that resets the **PATIENCE** counter. Consequently, this stochastic behavior leads to a prolonged training phase, eventually converging after 1,983 episodes.
- **Policy Loss Behavior:** A notable observation is the continuous oscillation of the policy loss. For us, this behavior was initially unusual, as our previous experience was limited to supervised learning where a decreasing loss curve is the primary indicator of success. However, in Reinforcement Learning, these oscillations are expected and correct; they reflect the logic of the policy gradient, where the loss is scaled by the reward. A high positive reward for a "good" action creates a large gradient spike to reinforce that behavior, and similar for the "bad" actions.



Figure 4: Reinforcement Training

Navigation Performance		Path Fidelity		Control Efficiency	
Metric	Value	Metric	Value	Metric	Value
Status	EXCELLENT	Mean CTE	0.0103 ± 0.0015 m	Avg Velocity	0.95 ± 0.01 m/s
Success Rate	100.0 %	Max CTE (Avg)	0.0282 m	Smoothness	0.0559 ± 0.0092
Steps (Avg)	295.1 ± 26.8	Tortuosity	1.000 ± 0.001	Avg Energy	15.15 ± 1.34

Table 2: Evaluation Metrics for the policy `v1_no_baseline`.

Evaluation Observations

Before deriving conclusions from the comparison between our trained policy and the Lyapunov baseline, we must ensure that the observed performance differences are statistically significant rather than the result of random variation. To this end, we implemented a statistical analysis module in `utils/statistics.py`.

This tool performs a **Wilcoxon signed-rank test** to calculate p -values, which tell us the statistical significance of the comparison. Additionally, it computes **Cliff's Delta** to quantify the effect size (magnitude).

Metric	Lyapunov	V1 No Baseline	P-Value	Cliff's Delta	Winner
Success Rate	100%	100%	1.00000	0.00 (Negligible)	Draw
CTE (Cross-Track)	0.0543 m	0.0103 m	0.00000 *	1.00 (Large)	<code>v1_no_baseline</code>
Smoothness	0.1158	0.0559	0.00000 *	0.96 (Large)	<code>v1_no_baseline</code>
Tortuosity	0.971	1.000	0.00000 *	-0.99 (Large)	<code>v1_no_baseline</code>
Steps (Avg)	414.7	295.1	0.00000 *	0.96 (Large)	<code>v1_no_baseline</code>
Energy	1.18	15.15	0.00000 *	-1.00 (Large)	Lyapunov

Table 3: Head-to-Head Statistical Analysis: Lyapunov vs. RL (`v1_no_baseline`)

The results demonstrate that the learned policy outperforms the Lyapunov baseline in path-following precision, as evidenced by the substantially lower Cross-Track Error (CTE) and a Tortuosity value of approximately 1.0. By optimizing for both higher average velocities and superior smoothness, the RL agent successfully reaches the goal in fewer steps than the analytical controller. However, this increased performance comes with a trade-off: the higher velocity profile and more aggressive tracking lead to a greater overall energy consumption compared to the conservative Lyapunov baseline.

3.6 REINFORCE Algorithm with Value Function Baseline

The next step was to update the previously implemented algorithm by introducing a **Value Function Baseline**. To reach it, the following architectural and procedural additions were implemented:

- **Value Network:** A second neural network was introduced. Its objective is to approximate the value function $V_\phi(s_t)$, which estimates the expected return-to-go from a given state. This network serves as the baseline for the policy update.
- **Modified Update Step:** The training iteration now incorporates two simultaneous optimization:

Value Update: Minimizes the MSE between $V_\phi(s_t)$ and the return G_t to improve baseline accuracy.

$$L_{val} = \frac{1}{N} \sum (V_\phi(s_t) - G_t)^2$$

Advantage (A_t): Quantifies how much an action's outcome deviates from the average expected behavior.

$$A_t = G_t - V_\phi(s_t)$$

Policy Loss: Re-centers gradients using A_t to reinforce only actions that exceed expectations.

$$L_{pol} = -\frac{1}{N} \sum (\log \pi_\theta \cdot A_t)$$

3.6.1 Value Network

The newly introduced **ValueNetwork** consists of a Multi-Layer Perceptron (MLP) with two hidden layers of 64 units each, utilizing *Tanh* activation functions. The final layer is a single linear scalar output representing $V(s)$.

To manage this second network, we introduced a specific hyperparameter: **VALUE_LR** (5×10^{-3}). This learning rate is set higher than the policy one because the advantage signal is only reliable if the baseline is accurate.

By forcing the value estimator to converge rapidly, we minimize the period during which the network provides "noisy" or incorrect advantage estimates to the policy.

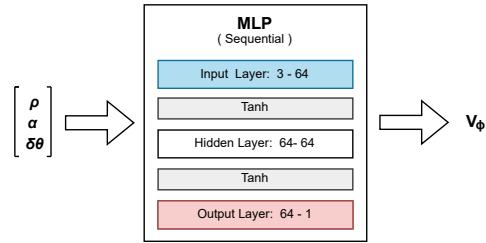


Figure 5: Value Network Schematic

3.6.2 Training Configuration and Results

To ensure a fair comparison, the training hyperparameters and random seeds were kept identical to the previous configuration. This consistency ensures the model faces the exact same environmental challenges as its predecessor, allowing us to isolate the impact of the Value Function Baseline.



Figure 6: Reinforcement Training

Training Observations

The training process was more stable compared to the previous version. This improvement resulted in reduced oscillations, which allowed the PATIENCE mechanism to trigger earlier and conclude the training more efficiently. Specifically, the agent converged in only 927 episodes; less than half of the 1,983 episodes required by the vanilla implementation. The only observed drawback was a slightly lower peak average reward compared to the previous model. To determine if this difference is significant we must examine the evaluation metrics and statistical comparisons.

Navigation Performance		Path Fidelity		Control Efficiency	
Metric	Value	Metric	Value	Metric	Value
Status	EXCELLENT	Mean CTE	0.0140 ± 0.0052 m	Avg Velocity	0.25 ± 0.06 m/s
Success Rate	100.0 %	Max CTE (Avg)	0.0291 m	Smoothness	0.0401 ± 0.0172
Steps (Avg)	353.4 ± 122.7	Tortuosity	0.998 ± 0.014	Avg Energy	2.23 ± 0.83

Table 4: Evaluation Metrics for the policy `v2_baseline`.

Metric	Lyapunov	V2 Baseline	P-Value	Cliff's Delta	Winner
Success Rate	100%	100%	1.00000	0.00 (Negligible)	Draw
CTE	0.0543 m	0.0140 m	0.00000 *	1.00 (Large)	v2_baseline
Smoothness	0.1158	0.0401	0.00000 *	1.00 (Large)	v2_baseline
Tortuosity	0.971	0.998	0.00000 *	-0.98 (Large)	v2_baseline
Steps (Avg)	414.7	353.4	0.00000 *	0.50 (Large)	v2_baseline
Energy	1.18	2.23	0.00000 *	-0.90 (Large)	Lyapunov

Table 5: Head-to-Head Statistical Analysis: Lyapunov vs. RL (`v2_baseline`)

Metric	V1 No Baseline	V2 Baseline	P-Value	Cliff's Delta	Winner
Success Rate	100.0 %	100.0 %	1.000	0.00 (Negligible)	Draw
CTE	0.0103 m	0.0140 m	0.051	-0.21 (Small)	Draw
Smoothness	0.0559	0.0401	0.00000 *	0.94 (Large)	v2_baseline
Tortuosity	1.000	0.998	0.198	0.12 (Negligible)	Draw
Steps (Avg)	295.1	353.4	0.00000 *	-0.96 (Large)	v1_no_baseline
Energy	15.15	2.23	0.00000 *	1.00 (Large)	v2_baseline

Table 6: Head-to-Head Statistical Analysis: RL `v1_no_baseline` vs. RL `v2_baseline`

Evaluation Observations

The comparison between the `v2_baseline` and the Lyapunov controller mirrors the results of the first implementation: the trained policy provides superior tracking precision and higher velocities at the cost of increased energy consumption.

More notably, the comparison between the two RL policies reveals that the shorter training time for v2 resulted in a negligible decrease in tracking performance. As the P-values indicate, these differences are not statistically significant, making the two models comparable in terms of path fidelity.

The primary distinction lies in the velocity profile: v1 learned a much more aggressive strategy. We hypothesize that because v1 underwent nearly double the training episodes, the velocity component of the reward function was optimized more extensively. In contrast, v2 reached stable tracking and smoothness within the first 1,000 episodes, suggesting that subsequent training in the vanilla model was primarily used to maximize velocity.

3.7 Final Comparison

Metric	Lyapunov	v1	v2
Success Rate	100.0%	100.0%	100.0%
Avg. Steps	414.7 ± 59.2	239.5 ± 34.2	376.1 ± 58.1
Mean CTE (m)	0.0543 ± 0.0089	0.0121 ± 0.0016	0.0134 ± 0.0029
Smoothness	0.1158 ± 0.0014	0.0650 ± 0.0216	0.0353 ± 0.0105
Tortuosity	0.9710 ± 0.0050	1.0001 ± 0.0043	0.9996 ± 0.0042
Avg. Vel (m/s)	0.0912 ± 0.0208	0.9444 ± 0.0088	0.2430 ± 0.0690
Avg. Energy (J)	1.1839 ± 0.1899	12.6686 ± 1.7494	2.1735 ± 0.6783

Table 7: Global Performance Comparison.

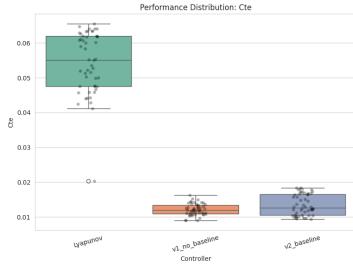


Figure 7: Boxplot Steps

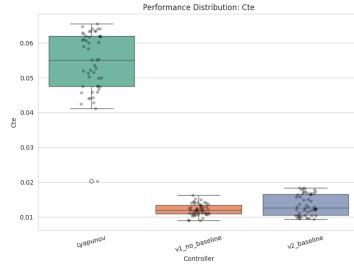


Figure 8: Boxplot CTE

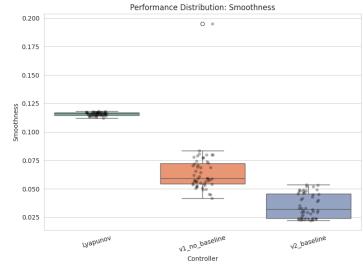


Figure 9: Boxplot Smoothness

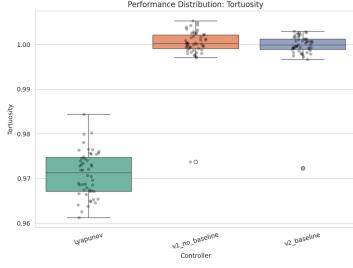


Figure 10: Boxplot Tortuosity

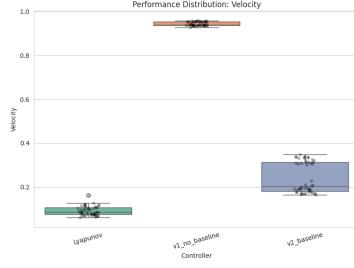


Figure 11: Boxplot Velocity

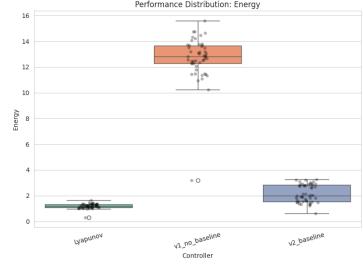


Figure 12: Boxplot Energy

Controller Videos: The demonstrations of the controllers in action are located in the `videos` folder within the submitted ZIP file. They are also available online at the following GitHub repository link:
https://github.com/orc-podavini-grisenti/final_assignment/tree/main/_documentations/tracking_videos

3.8 Future Works

Value Network Warmup: The fact that v2 does not show improved metrics over v1 leaves a sense that performance could be higher. While stable training generally suggests better results, this was not fully reflected in the final reward. To improve this, we propose a "Warmup" of the Value Network; by training the Value Network before the policy, we can reduce noisy behavior and high-variance updates at the start of the training process.

Lyapunov-Guided Advantages: During the implementation of the baseline, we hypothesized the potential benefits of using the Lyapunov Controller to provide the baseline for advantage estimates. Instead of initializing from a random baseline, incorporating a proven control law could anchor the advantages to physically consistent values. This would allow the model to learn more meaningful policy gradients from the very first iteration, leveraging existing control theory to guide the reinforcement learning process.

Exhaustive Hyperparameter Optimization: Due to limited time and computational resources, a full parameter tuning was not possible. A more exhaustive search (tuning learning rates, discount factors, and batch sizes) could lead to significantly different results and potentially unlock the full potential of the v2 architecture.

4 Navigation Policy

The goal of the second phase is to train a policy capable of autonomous navigation within the environment. The agent must successfully reach a specific target pose (position and orientation) while actively avoiding obstacles. To achieve this, the policy is trained using the **Proximal Policy Optimization** (PPO) algorithm.

4.1 Environment Update

Before working on the network, we must update the observation space of the environment. Previously, the observation space was 3-dimensional, consisting only of the egocentric error with respect to the goal. To enable autonomous navigation, we must now incorporate information regarding obstacles.

Our solution is to simulate a **LiDAR** system. We define 20 rays projecting from the front of the robot. The LiDAR simulator checks these lines to determine if an obstacle intersects them.

Each ray returns a distance value d_i ranging between 0 (obstacle attached to the robot) and `lidar_range` (the maximum LiDAR field of view, indicating no obstacle). Consequently, the final observation space increases to 23 dimensions:

$$[\rho, \alpha, \delta\theta, d_1, d_2, \dots, d_{20}]$$

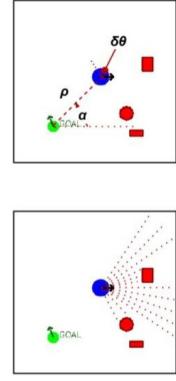


Figure 13:
Observation Space

4.2 PPO Algorithm

The Proximal Policy Optimization (PPO) algorithm was selected as the core learning mechanism for this navigation task due to its training stability, sample efficiency, and ease of tuning. Unlike standard Policy Gradient methods that are sensitive to step size, PPO ensures stable updates by avoiding updates that are too large with respect to the previous policy.

4.2.1 The Actor-Critic architecture

The agent utilizes an Actor-Critic framework, as illustrated in the architectural diagram (Figure 14). The Actor network is responsible for learning the policy $\pi_\theta(a|s)$, mapping the current observation state to a probability distribution over the continuous action space $[v, \omega]$ (linear and angular velocity). The Critic network estimates instead the Value Function $V_\phi(s)$, which predicts the expected cumulative reward from a given state. By using the Critic to calculate the Advantage $A_t = R_t - V(s_t)$, we effectively reduce the variance associated with policy gradients, allowing the Actor to learn which actions were better than average.

4.2.2 Clipped Surrogate Objective Function

The main idea behind the PPO algorithm is to perform small policy updates, small steps are more likely to converge to an optimal solution rather than bigger steps that can drive the agent out of path taking longer time or having no possibility to recover. PPO implements this logic with the so called **Clipped Surrogate Objective**. Instead of a standard log-likelihood loss, PPO calculates a ratio $r_t(\theta)$ between the new policy and the old policy. To prevent destructively large policy updates, the objective function "clips" this ratio within a range, typically $[1 - \epsilon, 1 + \epsilon]$ ($\epsilon = 0.2$ in our implementation). The final loss function is a composite of the clipped policy loss,

$$L_t^{CLIP} = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]$$

the Value Function (MSE) L_t^{VF} , and an Entropy Bonus to encourage exploration.

$$L_t = \hat{\mathbb{E}}_t \left[L_t^{CLIP} - c_1 \underbrace{(V_\theta(s_t) - V_t^{targ})^2}_{L_t^{VF}} + c_2 \underbrace{S[\pi_\theta](s_t)}_{\text{entropy bonus}} \right]$$

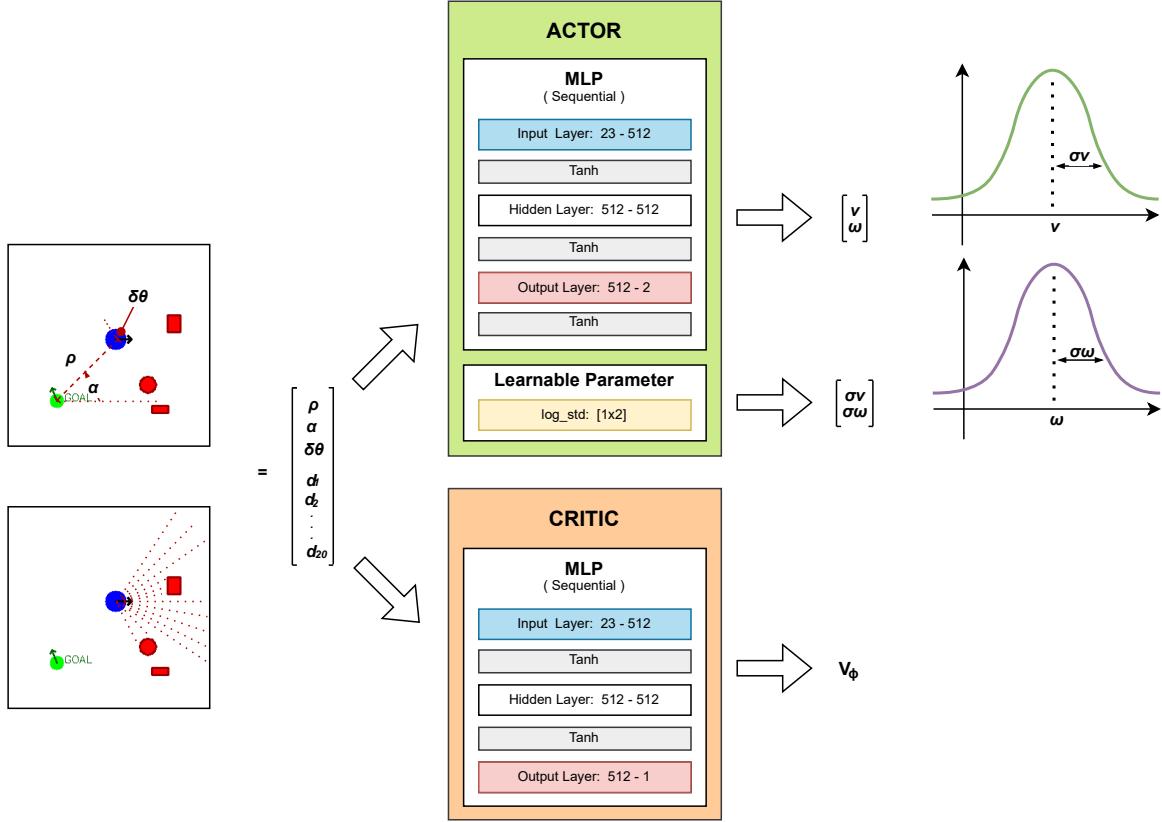


Figure 14: TTNetwork Architecture

4.2.3 Training Process and Policy Update

The training loop follows an "on-policy" collection strategy. The agent interacts with the environment for a fixed set of steps T , storing *observation-states*, *actions*, *log-probabilities* and *rewards* in a memory buffer. After the collection phase, the classic PPO algorithm compute the Advantage using Generalized Advantage Estimation (GAE), a temporal difference technique that helps stabilizing gradient updates and improving sample efficiency. In our case, we opted for a simpler Monte-Carlo approach, computing the empirical Advantage as:

$$\begin{aligned} G_t &= r_t + \gamma G_{t+1} \\ \hat{A}_t &= G_t - V_\phi(s_t) \end{aligned}$$

where:

- G_t is the discounted return at time t ,
- γ : The discount factor (GAMMA), which determines the present value of future rewards.
- r_{t+k} : The reward received k steps after time t .
- \hat{A}_t : The estimated advantage at time t .
- $V_\phi(s_t)$: The Value Function estimate (the Critic's output) for state s_t .

The model then performs several epochs of optimization on this batch of data using the Adam optimizer. This process iterates for a fixed number of episodes (10000 in our case) at which point the final Actor and Critic weights are exported for evaluation.

While we acknowledge that this method is more susceptible to noise in complex environments due to its higher variance, it provides an unbiased signal that proved sufficient for our navigation task. Preliminary results indicated that the agent successfully converged to a robust policy, achieving high success rates despite the simplified advantage estimation.

4.3 Reward Function Design

To guide the reinforcement learning agent through the complex task of both reaching a target and achieving a specific orientation, we implement a dense reward function. The total reward R_{total} is composed of a weighted sum of navigation and docking objectives, a safety component, and sparse terminal rewards.

4.3.1 Two-Zoned Reward logic

The task is divided into two phases based on the distance ρ to the goal. We define a transition factor τ to smoothly interpolate between a Navigation Zone (far from the goal) and a Docking Zone (near the goal), centered around a transition distance $d_{tr} = 1.5m$:

$$\tau = \text{clip}\left(1.0 - \frac{\rho}{d_{tr}}, 0, 1\right)$$

Based on this factor, we calculate dynamic weights for the two objectives:

- Docking Weight (w_{dock}): $0.2 + 0.6\tau$ (varies from 0.2 to 0.8)
- Navigation Weight (w_{nav}): $1.0 - w_{dock}$ (varies from 0.8 to 0.2)

This ensures that the agent focuses on global pathing when far away and gradually shifts focus toward precise orientation as it approaches the target.

4.3.2 Dense Rewards

The dense reward R_{dense} consists of four primary components, each normalized within the range $[-1, 1]$. Motion-based rewards (r_{closer} and the dynamic part of r_{obs}) are normalized by the maximum possible progress per step, defined as $\delta_{max} = v_{max} \cdot dt$, orientation-based rewards (r_{align} and r_{point}) are instead normalized thanks to the cosine function, that natively converts angles to a $[-1, 1]$ range. The following is a breakdown of all the reward components:

- **Alignment Reward (r_{align})**: Encourages the robot to approach the goal along a curve that matches the final required orientation:

$$r_{align} = \cos(\alpha - \Delta\theta)$$

- **Closer Reward (r_{closer})**: Rewards the reduction of Euclidean distance to the goal between time steps $t-1$ and t :

$$r_{closer} = \text{clip}\left(\frac{\rho_{t-1} - \rho_t}{v_{max} \cdot dt}, -1, 1\right)$$

- **Pointing Reward (r_{point})**: A geometric reward that is maximized when the robot's heading points directly at the goal:

$$r_{point} = \cos(\alpha)$$

- **Obstacle Reward** (r_{obs}): A safety term active when the minimum LiDAR reading d_L is below a warning threshold $d_w = 0.5m$. It combines a dynamic term (rewarding movement away from obstacles) and a static penalty:

$$r_{obs} = \text{clip}\left(\frac{d_{L,t} - d_{L,t-1}}{v_{max} \cdot dt}, -1, 1\right) - \frac{d_w - d_{L,t}}{d_w}$$

The total dense reward is calculated as follows:

$$R_{dense} = \underbrace{(r_{point} + r_{closer}) \cdot w_{nav}}_{\text{Navigation Phase}} + \underbrace{(r_{align} \cdot r_{closer}) \cdot w_{dock}}_{\text{Docking Phase}} + r_{obs} - 0.2$$

In the **Navigation Phase**, we use a *summation* because pointing and distance reduction are independent objectives; we want the agent to chase both simultaneously without complex interdependencies.

In the **Docking Phase**, we use *multiplication* because the relationship between alignment and movement direction is critical. The product ($r_{align} \cdot r_{closer}$) ensures:

- **Positive Reward:** If the robot approaches from the correct side (both terms positive) or if it moves away from the goal to "reset" its approach from an incorrect alignment (both terms negative).
- **Negative Penalty:** If the robot approaches the goal while incorrectly aligned (sign mismatch), discouraging "blind" forward movement.

Finally, the r_{obs} term is applied globally to maintain safety, and a constant step penalty of -0.2 is subtracted to encourage the agent to find the most time-efficient trajectory.

4.3.3 Terminal Rewards

To provide a strong signal for episode completion, sparse rewards are applied at the terminal state:

$$R_{terminal} = \begin{cases} W_{success} = +50 & \text{if goal reached (success)} \\ W_{collision} = -50 & \text{if robot collides} \\ W_{truncated} = -20 & \text{if time limit exceeded} \end{cases}$$

The total reward for any given step is the sum $R_{total} = R_{dense} + R_{terminal}$.

4.4 Evaluation

To assess the performance of the trained policy, we divide the evaluation into two distinct components:

1. **General Navigation Evaluation:** This provides a high-level overview of the navigator's general behavior and robustness across standard scenarios.
2. **Path Evaluation:** This focuses specifically on the geometric efficiency and the quality of the trajectory generated by the navigator.

4.4.1 General Evaluation

We evaluate the navigation performance within a training environment containing three obstacles. The policy is assessed using standard metrics, including Success Rate, Collision Rate, Mean Steps, Average Energy, and Path Length.

However, to better understand the agent's behavior regarding obstacle avoidance, we introduce the **Safety Margin**. This metric is particularly significant as it quantifies the agent's ability to maintain a secure buffer. The Safety Margin is defined as the mean of the minimum distances recorded between the agent and the nearest obstacle during each episode

Safety Performance		Navigation Efficiency	
Metric	Value	Metric	Value
Success Rate	96.0 %	Mean Steps	141.0 ± 183.1
Collision Rate	0.0 %	Avg Energy	7.52 ± 11.56
Safety Margin	0.929 ± 0.318 m	Path Length	2.90 ± 1.04 m

Table 8: General Navigation Evaluation for Training Environment (3 obstacles).

From the results, we observe that the combination of a high Safety Margin (0.929 m) and a **Collision Rate of 0.0%** indicates that the policy has effectively learned to maintain a secure buffer from hazards. The small percentage of failures is not attributed to collisions, but rather to episodes where the agent reached the maximum step limit before arriving at the goal.

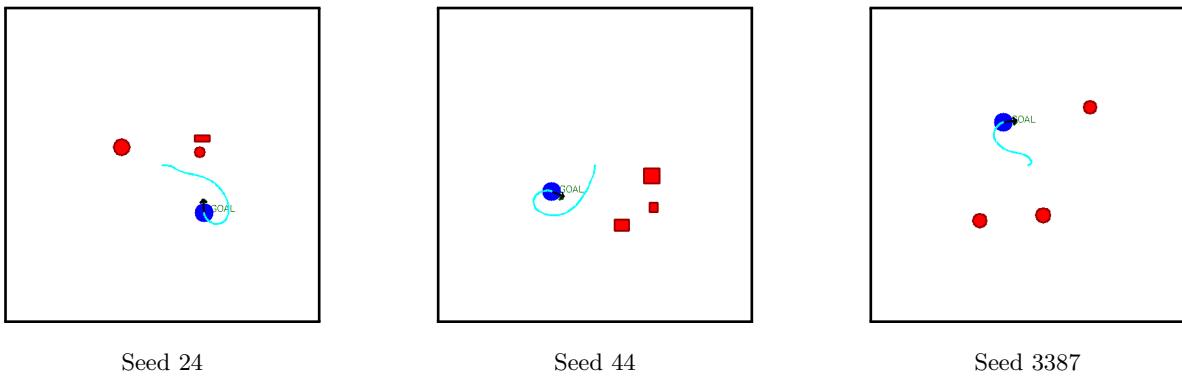


Figure 15: Visualizing different navigation behaviors

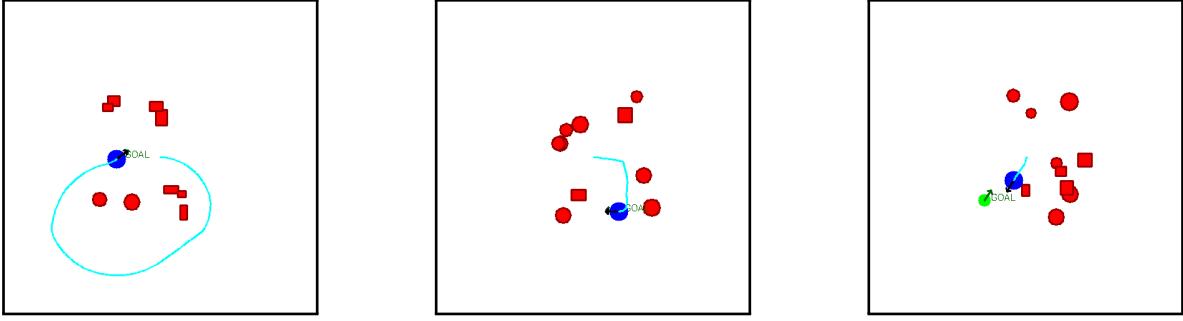
Stress Test: Complex Scenarios: To further assess the model’s limits, we tested the policy in a high-density environment featuring ten obstacles. This stress test evaluates the agent’s spatial reasoning and its ability to navigate through narrow passages.

Safety Performance		Navigation Efficiency	
Metric	Value	Metric	Value
Success Rate	90.0 %	Mean Steps	176.7 ± 164.4
Collision Rate	10.0 %	Avg Energy	9.21 ± 8.54
Safety Margin	0.670 ± 0.236 m	Path Length	3.37 ± 2.53 m

Table 9: Navigation Performance in a High-Density Environment (10 Obstacles).

The results remain robust even in these challenging conditions. As illustrated in Figure 16 (Seed 11 and Seed 56), the model exhibits a solid ability to circumvent obstacles and navigation in dense layouts. However, the 10% collision rate (e.g., Seed 23) highlights that the policy is not yet perfect.

Visual analysis of the episodes also reveals instances of “behavioral indecision,” where the agent moves significantly slower due to uncertainty in tight spaces. We anticipate that this hesitation can be mitigated through further reward function tuning to better penalize inactivity or low-velocity states in proximity to hazards.



Seed 11: Effective Avoidance

Seed 56: Dense Navigation

Seed 23: Collision Case

Figure 16: Navigation behaviors in complex scenarios with high obstacle density.

4.4.2 Path Evaluation

The second component of our analysis focuses on the geometric quality of the generated trajectories. We assess the navigator’s ability to plan paths that are not only safe but also efficient relative to a theoretical baseline.

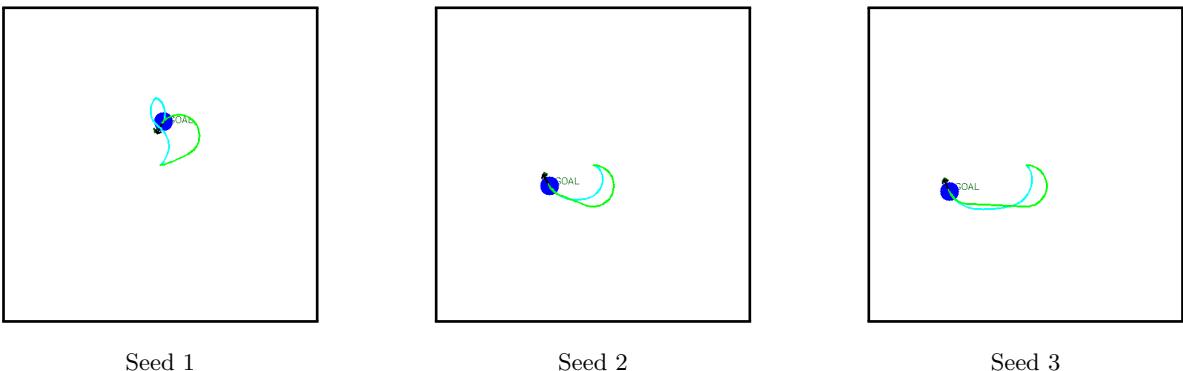
For this evaluation, **Path Efficiency** is defined as the ratio between the ideal path length; calculated using a Dubins path planner, and the actual distance traversed by the agent. It is important to note that this comparison is conducted in an obstacle-free environment, as our current Dubins planner implementation does not account for collision avoidance.

An average efficiency of **0.88** indicates that the policy generally travels a shorter distance than the ideal Dubins trajectory. This interesting result suggests that our model can outperform the Dubins baseline. We attribute this to the fact that the standard Dubins planner assumes a constant velocity, whereas our agent has the flexibility to accelerate and decelerate, allowing for more dynamic and tighter maneuvering.

While a more rigorous comparison would require implementing a more sophisticated kinodynamic planner, it was outside our time resources. Nevertheless, these results demonstrate that the paths chosen by our model are highly effective; in some instances, the agent closely mimics Dubins-like behavior, while in others, it adopts more optimized, non-linear trajectories.

Path Efficiency Metrics	
Metric	Value
Avg Ideal Length	3.9627 ± 1.2391 m
Avg Actual Length	3.2337 ± 1.1713 m
Avg Efficiency	0.8838 ± 0.4067

Table 10: Path Efficiency Analysis in Obstacle-Free Environments.



Seed 1

Seed 2

Seed 3

Figure 17: Visualizing path fidelity

Navigation Videos: The demonstrations of the navigation policy in action are located in the `videos` folder within the submitted ZIP file, both for the general evaluation and path evaluation. They are also available online at the following GitHub repository link:

https://github.com/orc-podavini-grisenti/final_assignment/tree/main/_documentations/navigation_videos

https://github.com/orc-podavini-grisenti/final_assignment/tree/main/_documentations/navigation_path_videos

4.5 Future Works

4.5.1 Algorithmic Refinements: GAE and Smoothness

The current implementation utilizes a Monte-Carlo approach for advantage estimation, which, despite producing satisfactory results, is subject to high variance in noisy environments. Future updates may focus on implementing Generalized Advantage Estimation (GAE) to balance the bias-variance tradeoff and to leverage all the feature of the complete PPO algorithm. GAE would in fact provide smoother gradient updates w.r.t. the current Monte-Carlo and Empirical Average approach.

This addition could help improve the smoothness of the robot's movements, allowing for more precise and less shaky trajectories. To further improve this aspect, the reward logic could be refined by implementing components that limit this behavior and fine-tuning the weights of the other components. Due to limited time and resources, it was not possible to thoroughly tune all the parameters.

4.5.2 Curriculum Learning Strategies

For our project, we used a fairly simple environment with few obstacles. One area that could be improved in the future is certainly the agent's ability to navigate more complex environments where obstacles are closer together and much more precise trajectories are required. To achieve this goal, a curriculum learning algorithm could be implemented, allowing the agent to learn in environments of increasing difficulty, building on the foundations acquired in previous stages. Some steps could be, for example, starting from an obstacle-free environment with the starting goal aligned with the starting point, gradually increasing the misalignment between the starting point and the goal, and finally adding obstacles, first fixed and then dynamic, thus increasingly restricting the agent's freedom of movement.

5 References

Our implementation of the Proximal Policy Optimization (PPO) algorithm is based on the following resources:

- **Hugging Face:** *Deep RL Course, Unit 8: Introduction to PPO*.
<https://huggingface.co/learn/deep-rl-course/unit8/introduction>
- **Medium:** *Mastering Proximal Policy Optimization (PPO) in Reinforcement Learning* by Felix Verstraete.
<https://medium.com/@felix.verstraete/mastering-proximal-policy-optimization-ppo-in-reinforcement-learning-230bbdb7e5e7>
- **Academic Paper:** *Implementation and Evaluation of PPO* by D. Bick (2021).
https://fse.studenttheses.ub.rug.nl/25709/1/mAI_2021_BickD.pdf

NB: Large Language Models (LLMs) were utilized as assistive tools for both the code implementation and the technical drafting of this report.