Benchmarking Continuous-Control Reinforcement Learning for Free-Floating Space Manipulator Trajectory Tracking and Quantifying the Impact of PPO Hyperparameter Optimization

Patrick Ermisch*
Faculty of Engineering and Computer Science, Frankfurt University of Applied Sciences

Abstract

Free-floating space manipulators exhibit strong dynamic coupling between manipulator motion and spacecraft base drift. This paper studies continuous-control reinforcement learning (RL) for end-effector trajectory tracking in a physics-based Simulink/Simscape environment with integrated safety monitoring. We benchmark six RL agents (TRPO, TD3, SAC, PPO, Policy Gradient, and DDPG) on the same circular end-effector tracking task and evaluate performance using a unified set of trajectory, stability, smoothness, energy, and safety KPIs. Across the benchmark, PPO achieves the best overall trade-off between tracking accuracy, base stabilization, training stability, and wall-clock efficiency. Building on this result, we quantify the effect of systematic PPO tuning (network architecture and key hyperparameters). Compared to a default PPO configuration, the optimized PPO improves the mean episodic return magnitude by 80.7%, reduces the mean squared end-effector tracking error by 84.9% and the maximum tracking error by 70.3%, and stabilizes the base orientation (72.2% reduction in orientation error). Moreover, the optimized PPO eliminates collisions and joint-limit violations in evaluation while slightly reducing energy consumption (15.98%).

Keywords: space robotics, free-floating manipulation, reinforcement learning, PPO, safe simulation, trajectory tracking

1 Introduction and Background

Free-flying robotic manipulators in space introduce unique control challenges compared to terrestrial robots. In orbit, a robot's base (typically a spacecraft) is not fixed. Any manipulator motion imparts an equal and opposite reaction on the base due to conservation of momentum. Consequently, moving an arm can unintentionally disturb the spacecraft's attitude or position, complicating precision tasks. Additional difficulties include communication latency and limited onboard computational resources, which further hinder real-time control and stabilization. Traditional model-based control approaches (e.g. reactionless planning or precise attitude control) require accurate system models and can struggle to adapt to uncertainties or unmodeled dynamics. These limitations motivate the exploration of learning-based control methods that can autonomously adapt to complex dynamics.

Reinforcement Learning (RL) has emerged as a promising approach for robotics, enabling agents to learn control policies through trial-and-error interactions. Recent advances in deep RL have shown success in complex control tasks, but applying RL to space robots is still relatively novel. The work presented in this paper investigates the use of continuous-action deep RL for a free-floating space manipulator, with a focus on systematically comparing different RL algorithms and ensuring safety through formal methods. The key contributions and novelties of this work include:

*steven.ermisch@stud.fra-uas.de

- Comparative Evaluation of RL Agents: We implement and evaluate six state-of-the-art continuous RL algorithms, Trust Region Policy Optimization (TRPO), Twin Delayed Deep Deterministic Policy Gradient (TD3), Soft Actor-Critic (SAC), Proximal Policy Optimization (PPO), vanilla Policy Gradient (PG), and Deep Deterministic Policy Gradient (DDPG), on an identical free-floating robot control task. This side-by-side comparison under unified conditions reveals which algorithm is best suited for controlling a free-floating space robot and why.

- Integration of Formal Safety Measures: To meet safety-critical space operation requirements, we incorporate formal methods into the learning loop. A collision monitor, momentum conservation check, and formal verification of joint torque limits are embedded in the simulation to enforce constraints and verify the learned policies. This integration ensures the RL agent cannot violate fundamental safety properties (like avoiding collisions or exceeding actuator limits) during training or execution.

- High-Fidelity Simulation Environment: We develop a high-fidelity MATLAB/Simulink simulation of a free-floating space robot, using a URDF-based multibody model and the Simscape physics engine. The robot, a cube-shaped base with a 4-DOF robotic arm, is modeled with realistic kinematics, dynamics, and sensor/actuator characteristics. A dedicated reinforcement learning environment provides the agent with observations and rewards, including a specially designed reward function that incentivizes accurate end-effector trajectory tracking while penalizing base movement, large joint velocities, and excessive control effort (energy). Through reward shaping and carefully chosen reset conditions, the agent learns to accomplish the task without causing undue base disturbance or safety violations.

- PPO Agent Optimization: Based on the comparative results, we identify PPO as the top-performing algorithm and further refine it for this task. We adjust the neural network architecture and tune hyperparameters (learning rate, batch size, clipping ratio, entropy regularization, etc.) to improve training stability and policy performance. The optimized PPO agent achieves significantly better tracking accuracy, smoother control inputs, and zero safety violations, outperforming the default configuration on all key metrics.

Overall, our study demonstrates that a combination of model-based simulation, deep RL, and formal verification can yield a robust and safe control policy for a free-floating space robot. In the following, we first describe the system modeling and RL framework, then detail the formal safety components, present a comparative analysis of the RL agents, and finally discuss the optimized PPO agent's performance, before concluding with broader insights and future directions.

2 System Modeling and Simulation Environment

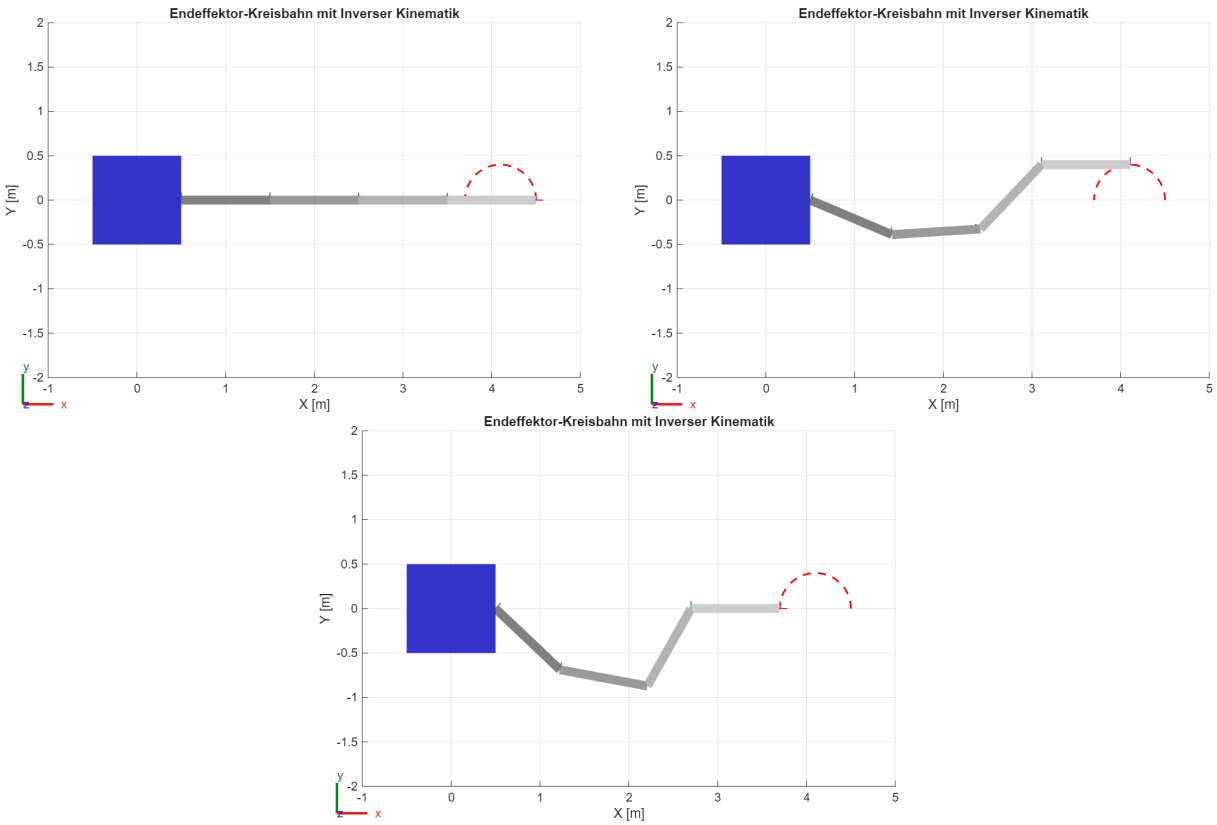


Figure 1: Validation of the desired end effector circular path using inverse kinematics

Robot Model: The testbed for this research is a simulated free-floating space manipulator inspired by the Spacecraft Robotics Toolkit (SPART) example. The robot consists of a cuboid base (serving as the spacecraft) with a 4-degree-of-freedom (4-DOF) serial manipulator arm mounted on it (see Figure 1). The arm has four revolute joints and four links, and its kinematic structure and inertial properties are defined in a Unified Robot Description Format (URDF) file (SpaceRobot.urdf) which is imported into Simulink's multibody simulation environment. Since SPART did not specify exact link geometries or masses, plausible values were chosen for this work and encoded in the URDF model. The result is a realistic multibody model with a floating base and an arm, capturing the strong dynamic coupling between them.

Dynamic Simulation in MATLAB/Simulink: The simulation is implemented in Simulink using the Simscape Multibody physics engine. Gravity is turned off (set to [0 0 0]) to emulate the microgravity orbital environment. The base is connected to the inertial world frame through an unactuated 6- DoF joint, allowing it to move freely in translation and rotation. No external forces or torques act on the system. Thus, the base only moves in response to forces generated by the manipulator's joints. This configuration ensures that momentum is conserved, any movement of the arm leads to a reaction on the base, as would occur in free space.

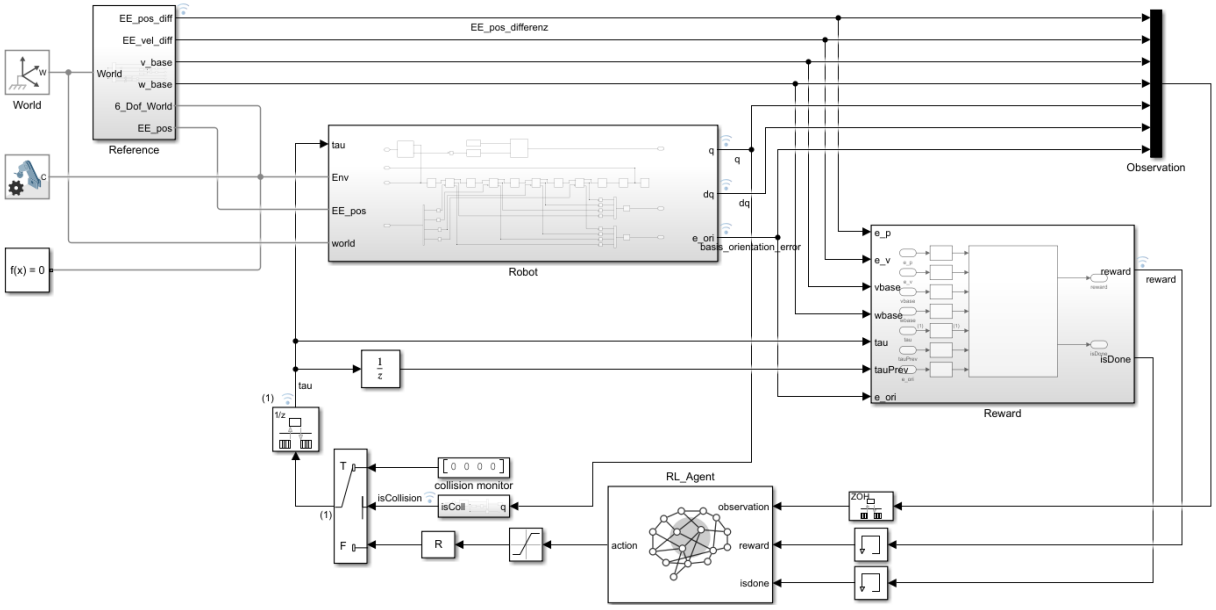


Figure 2: Overview of the Simulink model `SpaceRobot.slx`

The Simulink model is organized into modular subsystems, as illustrated schematically in Figure 2 (simulation architecture diagram). The main components of the model are:

- Robot Plant: a "Robot" subsystem that computes the full 6-DOF multibody dynamics of the free- floating robot (base + arm) at each time step. This subsystem takes joint torque inputs and outputs the resulting states (joint angles, base pose, velocities, etc.) by numerically integrating the equations of motion.

- Controller (RL Agent Block): a control subsystem where the learned RL policy is executed. The agent's output (joint torque commands) is applied to the robot through this block. Before reaching the plant, these commands pass through saturation and safety verification blocks to enforce actuator limits and safety constraints. For example, if the collision monitor (described below) signals an impending collision, this controller block will override all torques to zero, immediately bringing the system to a safe stop.

- Reference Trajectory Generator: a "Reference" subsystem that provides the desired trajectory for the robot's end-effector and (if needed) a desired base attitude. In this work, the reference is a circular path that the end-effector should follow in the inertial frame ("World"). The circular orbit's center, radius, and angular speed are parameterizable, allowing different difficulty levels. By default, the end-effector must trace a planar circle of a given radius at constant speed. The reference subsystem outputs the current target end-effector position (and velocity) at each time step, as well as a nominal desired base orientation (which in our case remains fixed, as we prefer the base to stay inertially stable).

- Sensors and Logging: various sensors measure the robot's state (joint angles, joint velocities, base orientation, base angular velocity, etc.), and their outputs are combined into an observation signal for the RL agent (detailed in the next section). Logging blocks record relevant data, and performance metrics are computed for evaluation. For instance, the instantaneous tracking error (difference between current end-effector position and reference) and measures of base motion are calculated continuously for use in the reward and for post-run analysis.

- Safety Monitors: crucial safety subsystems are integrated to monitor and enforce constraints. A Collision Monitor checks distances between the robot's components (e.g., between the

arm links and the base) and triggers a safety halt if any distance falls below a defined safe threshold d_{safe}. Similarly, a joint-limit monitor observes the arm's joint angles and triggers an emergency stop if any joint exceeds its allowed range. These monitors interface with both the plant and controller: they can terminate an episode and nullify the agent's torque outputs if a violation is detected (see Formal Methods Integration section for details).

To accurately simulate the continuous dynamics while interfacing with the discrete-time RL agent, a consistent integration scheme is used. The Simscape solver is configured to operate with a fixed-time step equal to the agent's decision interval (Δt), instead of the default variable-step, to ensure synchronization between the physics simulation and the RL decision-making cycle. Rate transition blocks handle any rate differences, avoiding aliasing and ensuring that state observations and action commands align correctly between the continuous plant and the discrete agent loop. Additionally, Simulink's unit consistency checks are enabled to catch any modeling errors (e.g. mismatched units) early in development.

Reference Trajectory and Task: The target task for the robot is to follow a periodic end-effector trajectory while minimizing base movement. As noted, the reference trajectory is chosen as a circular path (see Figure 1). Prior to training the RL agents, the trajectory was validated via inverse kinematics to ensure it is kinematically feasible for the manipulator (within joint limits and avoiding singularities). The inverse kinematics solution produced a consistent set of joint movements that achieve the desired end-effector motion, confirming that the circle can be tracked by the arm without requiring impossible motions. This precaution guarantees that the task is achievable, at least by some controller, so if the RL agent fails to learn it, it is due to learning limitations rather than an unattainable reference.

During simulation, the reference subsystem outputs both the desired end-effector position and velocity (as time series $EE_{ref}(t)$ and $EE_{vref}(t)$ in the workspace) at each time step. From these, instantaneous tracking error signals are computed: let e_p = (current EE position – desired position) and e_v = (current EE velocity – desired velocity). These error signals (position and velocity error of the end-effector) are key components of the observation and are also used in the reward computation (penalizing large errors) as described later. The trajectory parameters (circle radius, speed, etc.) can be varied to adjust the task's difficulty. For the experiments reported here, we choose moderate values that challenge the controller but do not push the arm to its dynamic limits.

In summary, the simulation environment provides a realistic and challenging platform for learning, with a free-floating base, a non-trivial trajectory control task, and integrated mechanisms to measure performance and enforce safety. Next, we describe how the reinforcement learning problem is formulated in this environment.

3 Reinforcement Learning Framework

State (Observation) Space: At each time step, the RL agent receives an observation capturing the system's key state information relevant to control. The state vector is composed of the following elements:

- End-Effector Tracking Error: The position error $e_p \in \mathbb{R}^3$ between the endeffector's current position and the reference target position (in Cartesian coordinates). If the end-effector is exactly on the desired trajectory point, e_p=0. Additionally, the error in end-effector velocity $e_v \in \mathbb{R}^3$ (difference between current and target endeffector velocities) is included. Together, (e_p, e_v) provides the agent a direct indication of tracking performance, it tells the agent how far off the trajectory the endeffector is and whether it is lagging or overshooting in velocity.

- Arm Joint States: The positions and velocities of each of the 4 arm joints. We include the joint angles $q \in \mathbb{R}^4$ and joint angular velocities $\dot{x} \in \mathbb{R}^4$ in the observation. These inform the agent of the manipulator's configuration and motion. (Note: Joint angles can be normalized or wrapped as needed, in our implementation they are provided in radians along with known joint limits.)

- Base Motion: The free-floating base's orientation and angular velocity. We parameterize the base's orientation by a three-parameter representation of the rotation (such as a normalized quaternion vector part or Euler angles) such that $ori_{base} \in \mathbb{R}^3$ represents the attitude error of the base from the reference (usually the identity rotation). Because the base is expected to remain mostly inertially fixed, this orientation error is ideally kept near zero by the agent's actions. The base's angular velocity $\omega_{base} \in \mathbb{R}^3$ is also included, as it captures any ongoing rotation of the base. (The base's translational velocity could also be considered, however, in a free-floating scenario with no external forces and zero initial momentum, the center-of-mass linear velocity remains constant. In our case the base's linear momentum is initially zero and remains near zero, so base translational motion is minimal by design. Thus, base linear velocity is not a major factor and can be omitted or included for completeness, our state focuses on rotational movement of the base.)

Combining these components, the overall observation vector has dimension $3(e_p) + 3(e_v) + 4(q) + 4(\dot{x}) + 3(ori_{base}) + 3(\omega_{base}) = 20 dimensions$ (in our implementation, a few additional state features like a boolean flag for terminal state can bring it to 23, but the core physical state is 20-dimensional). This rich state description provides the agent with all necessary information: how far the end-effector is from target, how the arm is configured/moving, and how the base is reacting.

Action Space: The agent's action is a vector of continuous joint torque commands for the manipulator's joints. At each decision step, the agent outputs $a = (\tau_1, \tau_2, \tau_3, \tau_4)$, corresponding to the torque to apply at each of the 4 revolute joints of the arm. These actions are real-valued and are constrained by the actuator capabilities. We enforce symmetric torque limits: each $|\tau_i| \leq \tau_{\max} (fori = 1..4)$. In the simulation, this is implemented by saturation blocks that clip the commanded torque to the range $[-\tau_{\max}, +\tau_{\max}]$. The value of $\tau_{\max}$ is set based on the robot's design (for example, $tau_{\max} = 5Nm$ for all joints, though the exact value is not critical as long as it is sufficient to follow the trajectory). These saturation limits not only model physical actuator limits but also serve as a safety measure to prevent excessive torques. The action space is thus continuous $\mathbb{R}_4$ (bounded), making this a continuous control problem that suits policy gradient and actor-critic RL methods.

The agent's policy (the controller we are learning) is a mapping from the 20-dimensional state space to the 4-dimensional action space: $\pi : s \mapsto a$. Internally, this policy is represented by a neural network whose parameters are adjusted during training to maximize cumulative reward (described next). The control frequency (agent decision frequency) is chosen such that the agent outputs a new torque command at a fixed interval $\Delta t (e.g., 0.1s)$. This Δt is the "agent sampling time" in Simulink and is consistent with the simulation's fixed step to maintain alignment.

Reward Function Design: A crucial aspect of the RL formulation is the reward signal, which guides the agent toward the desired behavior. We define the reward r_t at each time step t to reflect the dual objectives of accurate trajectory tracking and minimal base disturbance, along with penalties for unsafe actions. The reward is essentially the negative of a weighted cost function that includes several terms:

- End-Effector Tracking Error: We assign a high cost to deviations from the reference trajectory. For example, the instantaneous squared position error $||e_p||^2$ (and possibly velocity error $||e_v||^2$) can be used. This term drives the

- Base Motion Penalty: To keep the base as stable as possible, we include a penalty for base rotations and/or velocities. For instance, the squared norm of the base's angular velocity $||\omega_{base}||^2$ and orientation error $||ori_{base}||^2$ can be penalized. This term encourages the agent to execute arm motions in a way that minimizes imparted momentum to the base (in effect, preferring "reactionless" manipulation if possible).

- Joint Velocity and Torque Penalties: Very fast joint movements or sudden large torques are undesirable both for mechanical reasons and because they tend to disturb the base. We include a mild penalty on the magnitude of joint velocities $||\dot{x}||^2$and on control effort ($e.g., ||\tau||^2$). The latter also indirectly penalizes energy consumption. These terms ensure the agent prefers smoother, gentler motions rather than aggressive flailing, except as needed to improve tracking.

All these components are combined into an instantaneous cost function $c(t)$. The reward r_t is defined as a negative weighted sum of these cost terms, so that maximizing reward corresponds to minimizing the weighted costs. In formula form, one can express $r_t = -[W_p \parallel e_p \parallel^2 + W_v \parallel e_v \parallel^2 + W_b \parallel \omega_{base} \parallel^2 + W_{\dot{q}} \dot{x} \parallel^2 + W_\tau \parallel \tau \parallel^2]$, where the W coefficients are tunable weights that determine the relative importance of each term. We selected these weights empirically to balance the objectives (giving highest weight to end-effector error and base rotation, and smaller weights to smoothness/effort terms). The reward is thus higher (less negative) when the end-effector is on target and the base is steady, and lower (more negative) when the tracking error or base motion is large.

Reward Shaping and Terminal Rewards: In addition to the step-by-step reward, we implement several shaping strategies to aid learning:

- Persistent Tracking Bonus: To encourage the agent to not only reach the trajectory but stay on it, we give a small positive reward for each time step the end-effector remains very close to the reference (within a tolerance). This is equivalent to lowering the cost when $\|e_p\|$ is below a threshold, and helps reinforce good behavior over extended durations.

- Terminal Success Reward: If the agent successfully completes an entire episode following the trajectory (e.g., completes one full circle) without any safety violation, a one-time bonus reward R_{goal} is given at the end of the episode. This emphasizes the final objective of finishing the task. We adjusted R_{goal} such that successful episodes are significantly rewarded, encouraging the agent to complete the trajectory rather than stopping early.

- Safety Violation Penalty: Conversely, if at any time a safety-critical event occurs, such as a collision or a joint exceeding its limit, the episode is immediately terminated and a large penalty is applied. In our implementation, any such violation triggers an immediate reward of -1 (a very negative value in our normalized reward scale) and the episode ends. This teaches the agent to avoid those catastrophic actions. Similarly, if the system state diverged (e.g. due to a numerical instability or an "explosion" of the controller), we also terminate and penalize heavily to prevent the agent from exploring dangerous, unphysical regimes.

- Numerical Guard: As a precaution, if the end-effector strays extremely far from the reference (indicating the agent might be doing something nonsensical, or a simulation error), we abort the episode with a heavy penalty. Specifically, if the end-effector position error exceeds a very large threshold (e.g. >10 units, far outside any reasonable range), we consider it a fault and end the trial. This prevents rare out-of-bound states from dominating training statistics.

These shaping measures were implemented via a reward function script in MATLAB (embedded in a Simulink MATLAB Function block). The weights of each term (stored in a structure W)

were tuned iteratively to ensure the agent receives informative feedback. The goal was to make the learning process efficient by guiding the agent away from obviously bad actions (crashing the robot) and towards the desired behavior, while still allowing enough exploration to discover the optimal policy.

Training Procedure: We adopt an episodic training approach. Each episode corresponds to a simulation of fixed duration (a finite horizon H seconds or a fixed number of time steps N). In our case, one episode lasts long enough for the end-effector to attempt following the circular trajectory (e.g., one or two loops of the circle). At the start of each episode, the system is reset to an initial condition by a custom reset function. The initial joint angles are set to a feasible configuration (e.g., the arm in a certain pose near the start of the trajectory) and the base is at a defined attitude (often the identity orientation, i.e., no initial rotation) or a small random perturbation thereof. All state variables, observer integrators, and logging signals are reset. We ensure the initial state is within safe bounds (no collisions or limit violations at t=0). The reset function can randomize the starting joint angles within a range or the phase of the trajectory to improve generalization, though within this study, we kept a consistent start pose for simplicity (with the end-effector at a specific point on the circle).

Once an episode begins, the agent receives observations at each step and outputs torque actions. The simulation runs until either the time horizon is reached (the episode length H) or a terminal condition occurs (task completed or safety violation). As described, reaching the end of the trajectory successfully or hitting a failure condition will terminate the episode early. In practice, an episode might end early if, for example, the agent causes a collision at 70% of the trajectory, it will receive the penalty and the episode stops there.

We use a standard policy gradient training loop (as provided by MATLAB's RL toolbox): after each episode (or after a set number of steps), the collected state, action, reward trajectories are used to update the agent's policy (and value function, if applicable) via the chosen RL algorithm. The specifics of each algorithm (PPO, DDPG, etc.) are handled by their respective implementation, we supply the experience and let the algorithm update the neural network. All agents were trained for the same number of episodes to allow fair comparison. We chose $N_{train} = 1000$ episodes for each agent, which was sufficient for most algorithms to converge or reach a performance plateau. During training, we monitored metrics such as the cumulative reward per episode, the end-effector tracking error, and the base motion, averaged over each episode. A successful training run is characterized by the episode reward increasing (toward 0, since negative costs are used) and the tracking error and base movement decreasing over time. We also tracked the number of episodes that ended prematurely due to safety triggers, aiming for this to go to zero as training progresses.

For each algorithm, we performed multiple training runs and selected the best-performing policy (based on highest average reward achieved) for final evaluation. In practice, we found relatively consistent results across runs for the more stable algorithms (PPO, TRPO), whereas some runs of less stable methods (e.g. DDPG) would diverge, underscoring the importance of stability.

Evaluation and Metrics: After training, we evaluated each agent's performance on $N_{eval} = 30$ independent episodes (with various initial conditions) to gather statistics. Over these evaluation runs, we compute a set of Key Performance Indicators (KPIs) to quantitatively compare the agents. These KPIs include:

- K1: Average Episodic Return, the mean cumulative reward per episode (higher is better, this reflects overall success in the task).

- K2: Mean Squared End-Effector Position Error, averaged over the episode (lower is better, measures tracking accuracy).
- K3: Maximum End-Effector Position Error, worst-case deviation during the episode (lower is better).
- K4: Mean Base Orientation Error, e.g., average quaternion error magnitude (in radians) indicating how far the base drifted in attitude (lower is better).
- K5: Collision Rate, the fraction of episodes in which a collision occurred (0 means no collisions, we expect this to be zero if the agent behaves safely).
- K6: Joint Limit Violation Rate, fraction of episodes where any joint exceeded its range (again, 0 is ideal).
- K7: Control Smoothness (Jerk), a measure of actuation smoothness, computed as the normalized jerk (time-derivative of acceleration) or second difference of joint torques. Lower K7 indicates s
- K8: Mean Base Angular Velocity, a measure of residual base rotation (deg/s or rad/s) during the task (lower is better, ideally zero).
- K9: Energy Consumption, approximate average energy expended by the joints (e.g., $\int(\tau \cdot \omega)dt$, in some units) or a proxy like sum of squared torques over time. Lower energy usage is better if achieved without sacrificing performance.

Additionally, we consider training-related metrics for comparison: T1: variance of the episodic reward in the later stage of training (indicating training stability), T2: variance of the value function or average reward (another stability metric), and T3: training time required to reach 1000 episodes (reflecting sample efficiency and computational load). These help to evaluate how easily an algorithm learns in this environment.

By evaluating all agents on these common metrics, we obtain an objective comparison of their performance and characteristics. The next section presents and discusses these comparative results.

4 Formal Methods Integration in the Learning Loop

Safety is paramount for space robotics. An autonomous controller must respect physical constraints and avoid dangerous behaviors, especially when learning is involved. In this work, we integrate formal safety mechanisms directly into the RL training and execution environment to ensure the learned policies are not only effective but also safe and physically valid. The following formal methods and safety components were used.

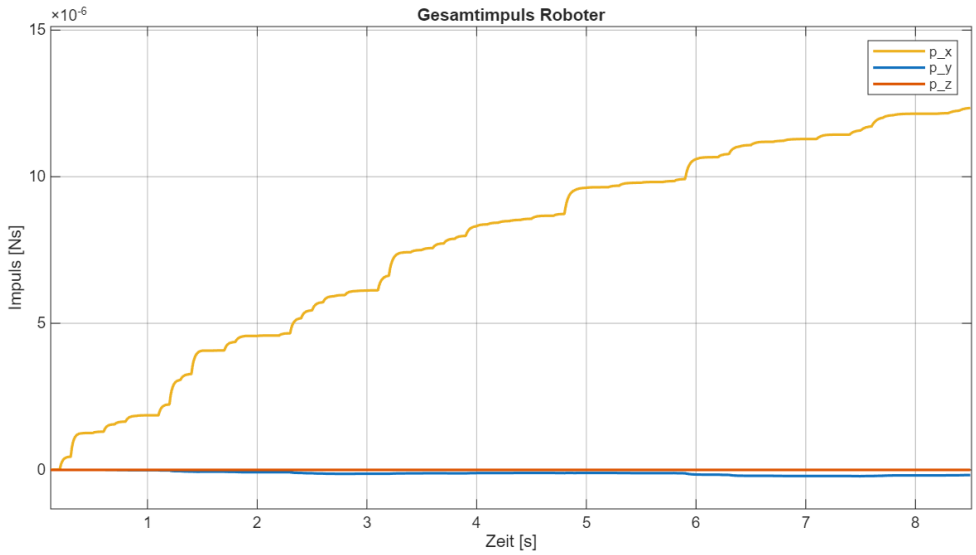
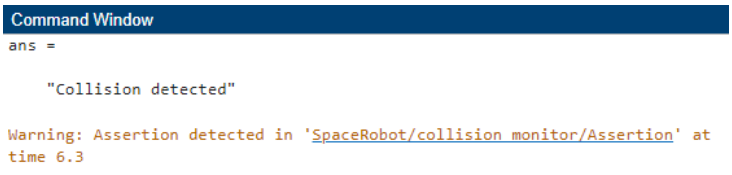
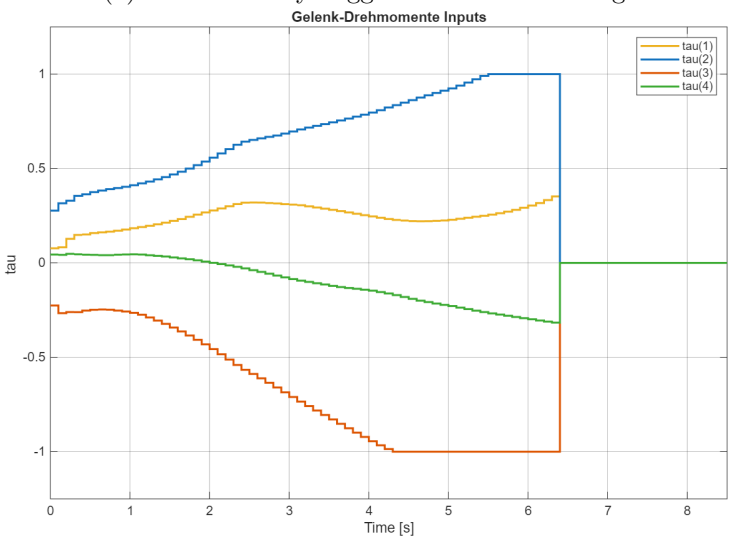


Figure 3: Course of the robot's total momentum

Momentum Conservation Monitor: In a free-floating system without external forces, the total linear momentum of the robot (base + arm) should remain constant (and in our case, near zero since initial momentum is zero). We implemented a Momentum Observer that continuously calculates the total linear momentum p_{tot} of the system and checks for conservation. This is done by attaching virtual sensors to each body (base and links) to measure their velocities and computing $p_{tot} = \sum_i m_i v_i$. In an ideal simulation, p_{tot} stays roughly zero at all times (small numerical deviations on the order of 10^{-6} Ns may occur due to integration errors). If a significant, systematic change in p_{tot} were detected, it would indicate an unintended external force or a modeling error (such as an unmodeled constraint or bug). During our experiments, p_{tot} remained near 0 (within 10^{-6} Ns) throughout each episode with the learned policy, confirming physical consistency (see Figure 3). This momentum check provides confidence that the simulation adheres to Newton's laws and the agent isn't exploiting any unphysical loophole. It also serves as a diagnostic tool: any momentum drift would pause training for model inspection



(a) Automatically triggered collision warning



(b) Scope of torque inputs in the event of a collision

Figure 4: Representation of collision detection and the associated torque curves

Collision Monitor and Avoidance: We define a minimum allowable distance between certain parts of the robot (for example, between the end-effector and the base structure). A Collision Monitor subsystem computes distances between predefined pairs of bodies each step (using the geometry from the URDF and Simscape's collision detection capabilities). If any distance falls below the safe threshold d_{safe} (indicating a potential collision or self-collision is imminent), the system triggers a collision event. In our simulation, we set, e.g., d_{safe}=5 cm as the minimum clearance. When triggered, the following occurs: (1) a warning flag is raised and logged (for analysis), (2) a done signal is sent to terminate the episode immediately, and (3) all joint torque commands from the agent are overridden to zero to haltmotion. This mechanism effectively acts as an emergency brake, preventing the agent from continuing once a collision is about to happen. The collision monitor was tested by deliberately commanding the arm toward the base. It reliably detected the breach of distance and stopped the system (see Figure 4). During training of the RL agents, collisions occurred in early exploratory episodes for some agents, but the monitor caught them and ended those episodes with heavy penalties. As training progressed, all agents learned to avoid collisions altogether, indeed, in the final evaluation all agents achieved K5 = 0 collisions per episode, confirming the efficacy of this safety measure.

Joint Limit Enforcement: The robot arm has joint angle limits (e.g., each revolute joint might be limited to ±180° or a smaller range based on mechanical constraints). Violating joint limits could damage a real robot and often indicates an unstable controller. We enforce joint limits at two levels. First, in the Simscape model, we include mechanical joint limits with penalty forces (a stiff barrier that resists further motion beyond the limit). Second, we add an Assertion block that monitors each joint angle. If any joint exceeds its predefined limit, the assertion triggers a violation event. Similarly to the collision case, we handle a joint limit violation by: terminating the episode, applying a large negative reward, and zeroing out all torques immediately. In fact, the Simulink model is configured such that when the assertion triggers, it directly sets the torque

inputs τ to zero in that same time step as a protective measure. This ensures the robot doesn't push further into the limit or oscillate. In our results, we observed very few limit violations once agents were trained. Among all algorithms, PPO and TRPO had a handful of episodes grazing the limits (K6 values of a few percent), whereas TD3, SAC, PG, and DDPG essentially never hit the limits ($K6 \approx 0$). Notably, PPO's superior tracking performance came at the cost of occasionally using the full joint range (K6 = 0.0505, i.e. ~5% of episodes, had minor limit violations). Thanks to the joint limit enforcement, these events did not lead to instability: the agent's torques were cut off momentarily when a limit was reached, preventing any damaging behavior, and the episode was marked as failed. Over time, the agent learned to avoid this scenario as well, especially in the optimized PPO version (where K6 dropped to 0).

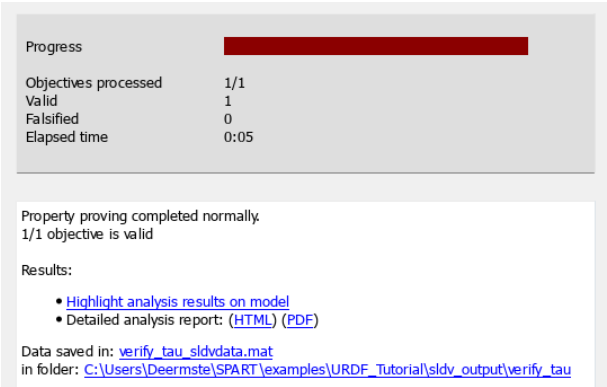


Figure 5: Result of the Simulink Design Verifier

Formal Verification of Torque Constraints: In addition to enforcing torque saturation during simulation, we performed an offline formal verification to prove that the trained policy could not command out-of-bounds torques. Using MATLAB's Simulink Design Verifier (SLDV) in propertyproving mode, we analyzed a simplified closed-loop model of the agent controlling the robot's joints. The property to prove was that for all possible agent inputs (observations), each output torque τ_i remains within $[-\tau_{\max}, +\tau_{\max}]$. The verification succeeded: SLDV was able to prove that no counterexample exists and the torque limits are always respected by the policy network. Figure 5 shows the SLDV result confirming the property (no violations found). This formal assurance is important because it considers all possible states, including ones not seen during simulation, and guarantees the policy cannot output a dangerous torque. It's worth noting that directly running formal analysis on the full Simscape model was not feasible, the nonlinear dynamics are too complex for the automated theorem proving. To circumvent this, we extracted the relevant part of the model (the control law and saturation logic) into a separate simplified model (Verify_tau.slx) where the plant was replaced by a worst-case abstraction (essentially, we just needed to check the policy output block). This exemplifies how formal methods can complement simulationbased learning: by verifying critical properties on an abstract model of the agent, we gain additional confidence in safety beyond empirical testing.

Limitations of Formal Methods: While the above components significantly enhance safety, we acknowledge that they cover a limited set of properties. They ensure zero collisions, no joint overextensions, and bounded torques, and they monitor physical consistency (momentum). However, other aspects, such as guaranteeing task completion within a time or stability margins, were not formally verified. A complete formal specification of the entire mission (for instance, something like temporal logic specifications of "the robot shall accomplish X without Y happening") would be far more complex. Our approach selectively targets the most critical safety concerns. In practice, this proved sufficient to train successful policies without a single catastrophic failure: episodes that would have been catastrophic were safely stopped by the

monitors (and heavily penalized to teach the agent). As a result, by the end of training, the learned agent's behavior was entirely within the safe envelope enforced by the formal constraints.

In summary, integrating these formal safety checks into the learning process was vital. It not only protected the simulation (and a potential future real system) from damage during the exploratory phase of learning, but it also shaped the agent's behavior by penalizing unsafe actions. The agent thus learns with a form of built-in "safety shield." This approach demonstrates a viable path to apply deep RL in safety-critical domains like space robotics, where pure trial-and-error without safeguards would be unacceptable. We next analyze the performance outcomes of the various RL agents under these conditions.

5 Comparison of Continuous RL Agents

A central question of this study is: Which continuous-action RL algorithm is most effective for controlling a free-floating space robot? To answer this, we evaluated six algorithms from three major families on the same task and environment. The algorithms compared are:

- Policy Gradient (PG): the classic REINFORCE algorithm (Monte Carlo policy gradient), included as a baseline for direct policy optimization without advanced variance reduction or stability t
- Proximal Policy Optimization (PPO): a modern policy-gradient method that uses clipped probability ratio updates to ensure stable and conservative policy improvements. PPO is known for its balance of stability and performance.
- Trust Region Policy Optimization (TRPO): a policy-gradient method that enforces a trust-region constraint (via KL-divergence) on policy updates. TRPO was expected to perform similarly to PPO in terms of stability, albeit with higher computational cost due to solving a constrained optimization each update.
- Deep Deterministic Policy Gradient (DDPG): an off-policy actor-critic algorithm with a deterministic policy and target networks. DDPG can excel in continuous control by learning a Qfunction and a policy simultaneously, but it is known to be sensitive to hyperparameters and can be unstable.
- Twin Delayed DDPG (TD3): an improved variant of DDPG that addresses function approximation issues by using two critic networks to mitigate overestimation bias and by delaying policy updates. TD3 generally offers more stable learning than standard DDPG.
- Soft Actor-Critic (SAC): an off-policy actor-critic algorithm that learns a stochastic policy and includes an entropy term in the reward to encourage exploration. SAC aims to achieve both high reward and high entropy, making the agent's behavior robust and exploratory. It often achieves very good performance but can be computationally heavier due to sampling and entropy calculations.

These algorithms cover a spectrum from on-policy to off-policy, from deterministic to stochastic, and from simpler to more complex update rules. Based on prior knowledge and the nature of our problem, we hypothesized that algorithms with more robust, stable policy updates (like PPO and TRPO) would likely outperform others in this highly coupled, continuous control scenario. Off-policy methods (DDPG, TD3, SAC) might reach good performance as well, but could suffer from tuning sensitivity and stability issues given the complex dynamics and safety constraints. Indeed, DDPG and basic PG were expected to be the least reliable due to known tendencies for instability without additional regularization.

Training Setup: Each agent was trained under identical conditions for fairness. We used the same network architecture (initially two hidden layers of 128 units each for policy and for value function, where applicable) and same reward structure for all. Hyperparameters (learning rate, discount factor $\lambda = 0.99$, etc.) were initially set to default reasonable values and kept consistent across agents as much as possible. This "apples-to-apples" approach means any performance differences can be attributed primarily to the algorithmic differences, not to unequal tuning. (Later we will tune PPO further, but for the main comparison, PPO uses a default configuration as well.) Each agent's training run of 1000 episodes yielded a final policy which we then evaluated.

Performance Metrics: We evaluated the agents on the KPIs described earlier (tracking error, base stability, etc.) by running 30 test episodes for each. Table 1 below summarizes a subset of these metrics, highlighting the differences between agents:

KPI (unit)	**TRPO**	**TD3**	**SAC**	**PPO**	**PG**	**DDPG**
K2: Mean EE position error (m^2)	0.0680	0.1914	0.1908	**0.0257**	0.1638	0.0937
K3: Max EE error (m^2)	0.6783	0.5341	0.7766	0.4833	1.0654	**0.4758**
K4: Mean base orientation error (−)	0.0694	0.0868	0.1687	**0.0667**	0.1302	0.2682
K6: Joint limit violation rate	0.0214 (2.1%)	0	0	0.0505 (5.1%)	0	0
K7: Torque smoothness (norm. jerk)	0.7498	0.9981	0.7536	**0.6812**	0.7125	0.8118
T1: Reward std. (late training)	**0.6144**	8.5168	9.6907	1.7756	13.1970	64.4764
T3: Training time (1000 eps)	~13 min	~38 min	~35 min	**~8 min**	~8 min	~24 min

Table 1: Selected performance metrics for each RL agent (best values in bold). PPO excels in most categories, achieving the lowest tracking errors (K2, K3), best base stability (K4), and smoothest control (K7), while maintaining short training time. TRPO is a close second in many metrics but requires more training time. Off-policy methods (TD3, SAC, DDPG) show larger errors or instabilities (e.g., high T1 for DDPG) despite some strengths (DDPG's low max error K3). (KPI data from evaluation of 30 episodes per agent.)

From these results, several observations can be made:

- Training Stability: TRPO and PPO had the most stable training process, as reflected by the very low variability in their late-stage episodic rewards (T1). PPO in particular converged smoothly. Its reward variance was an order of magnitude smaller than that of SAC or TD3. By contrast, DDPG and (to a lesser extent) PG showed highly erratic training behavior (DDPG's reward variance T1 was extremely high), often oscillating or diverging without careful tuning . This indicates that on-policy methods with robust policy updates (PPO/TRPO) are better suited to this problem, likely because the environment is complex and sensitive, so stable updates prevent the policy from collapsing under bad data.

- Sample Efficiency and Speed: In terms of wall-clock training time (T3), PPO was the fastest to train, completing 1000 episodes in around 7–8 minutes on our machine. This

efficiency is partly due to PPO's ability to learn effectively from fewer, on-policy samples and its simpler update per iteration. PG (which is essentially a simpler version of PPO without a critic or clipping) also was quick, but it didn't learn a good policy. Off-policy methods like TD3 and SAC took substantially longer (~35–38 minutes), owing to their more complex updates and potentially more steps required for convergence. TRPO was moderate (13 minutes), faster than SAC/ TD3 but slower than PPO, which is expected given TRPO's heavier computation per step. Thus, PPO and TRPO not only learn stably but also relatively quickly, whereas SAC/TD3 demand more computation (likely due to gradient computations for critics and larger replay buffer sample requirements).

- Trajectory Tracking Performance: PPO achieved the best tracking accuracy among all agents. Its mean squared end-effector error K2 was only 0.0257 (in normalized units), significantly lower (better) than the next-best (TRPO at 0.0680). This means PPO's learned policy kept the end-effector very close to the desired path most of the time. The maximum error K3 for PPO was also very low (0.48, roughly tied with DDPG's 0.47 as the smallest). DDPG's low max error is interesting, it suggests that in some successful runs, DDPG can track extremely well when it works. However, DDPG's poor stability means it doesn't achieve that consistently. TRPO had slightly higher tracking error than PPO (still good), whereas TD3, SAC, and PG showed notably larger errors (e.g. PG's mean error was ~6 times PPO's, indicating it struggled to keep on the trajectory). Qualitatively, we observed that PG's policy oscillated and often missed the trajectory, while SAC and TD3 sometimes followed parts of the trajectory but with larger deviations, possibly because they never fully mastered the coupling dynamics. In short, PPO's precision in trajectory following was unmatched in this set, with TRPO a solid second place.

- Base Motion Minimization: Keeping the base (spacecraft) stable is a critical objective. Here again PPO and TRPO excelled. The average base orientation error K4 for PPO was 0.0667 (in quaternion units, implying the base stayed within a few degrees of its initial attitude on average) . TRPO was comparable at 0.0694. In contrast, DDPG's base orientation error was 0.2682, much larger, indicating that DDPG's policy allowed the base to drift or rotate significantly. SAC was also relatively high (0.1687), and even PG was higher (0.1302) than PPO. Interestingly, TD3 achieved a very low base angular velocity (as per K8 ~0.02 rad/s, better than others which were ~0.08–0.1), suggesting TD3's policy might have learned to almost freeze the base, likely by hardly moving at all, as evidenced by its poor tracking accuracy. Indeed, TD3's strategy seemed to be overly conservative, moving slowly so as not to disturb the base, but then failing to track the trajectory accurately. All agents maintained zero or near-zero base translation (the center of mass remained essentially fixed as expected due to momentum conservation). The key takeaway is that PPO and TRPO managed to minimize base rotations while still accomplishing the task, whereas others faced a trade-off or simply did poorly in one aspect (e.g., DDPG achieved low max EE error at the expense of base stability).

- Safety and Constraint Violations: Thanks to the integrated safety mechanisms, none of the agents caused irrecoverable failures, but they did vary in how often they hit the enforced limits. All agents successfully avoided any collisions (K5 = 0) during the evaluation runs. This indicates the collision avoidance strategy was effective across the board, a success of the training with penalties and the monitor. For joint limits (K6), as noted, PPO had the highest incidence of limit approaches: about 5% of episodes saw a joint touch its limit. TRPO had ~2% of episodes with a limit event. The others were essentially zero. This suggests PPO (and TRPO) were pushing the arm to its workspace extremes to achieve the best tracking, whereas the others, being less precise or aggressive, stayed well within the joint ranges. The result itself is not alarming, a few limit touches, handled gracefully by the safety logic, in exchange for better tracking. It does highlight an interesting point: the

best tracking agent (PPO) uses more of the available workspace and occasionally bumps against the constraints. In a real system, this might be acceptable if the constraints are soft, but it also indicates that further reward tuning or slight modifications could teach PPO to leave a safety margin.

- Control Smoothness: We quantified smoothness via the jerk-based metric K7. PPO again was best (lowest jerk), meaning its torque commands changed most gradually on average. This is an important practical quality: smooth torques imply less excitation of unmodeled dynamics (like flexible modes) and less wear on motors. TRPO, SAC, and PG were slightly higher but in a similar range as PPO, while TD3 and DDPG produced noticeably jerkier control signals (their K7 were highest, ~0.8–1.0). This aligns with anecdotal observations that DDPG/TD3 policies tended to thrash more, possibly a side effect of their less stable learning, resulting in noisier policies. The fact that PPO yields the smoothest control is another point in its favor for real-world implementation.

- Energy Usage: Although not as critical as other metrics, energy (K9) differences were observed. TRPO and PG had the lowest energy consumption during the task (TRPO K9 ~3.76, PG ~5.33, in arbitrary units). PPO's energy use was moderate (~6.33), while SAC was highest (~9.09) and DDPG relatively low (~0.82). The extremely low values for DDPG and TD3 (0.8185 and 0.0893 respectively) suggest those agents were not driving the system much at all, likely failing to track the trajectory vigorously (hence low energy, but also high error). Conversely, SAC's high energy means it was applying a lot of effort, perhaps inefficiently. PPO used more energy than TRPO because it moved faster to get better tracking, but still remained within reasonable bounds. In real missions, minimizing energy is important, but there is usually a trade-off with performance. PPO struck a balance: it used more energy than the minimum, but delivered top performance. Notably, after PPO optimization (discussed next), its energy usage actually drops while performance improves, indicating we can have both better tracking and lower energy with a well-tuned agent.

In qualitative terms, watching the simulations: the PPO agent clearly demonstrated the best behavior, the arm moved swiftly and accurately along the circle, while the base barely rotated (one could see slight counter-movements which quickly damped out). The TRPO agent was also quite competent, but with slightly slower response and occasionally the base drifted a tiny bit more (still small). SAC managed the task but with more overshoot. It tended to oscillate around the path and the base had some jitter (perhaps due to exploratory noise from entropy maximization). TD3 often under-acted, the arm would start following the circle but lag behind, seemingly afraid to induce base motion, leading to large steady-state errors. DDPG was inconsistent: some trials it would nearly complete the circle with good precision (hence the low max error in one of the best runs), other times it would destabilize and flail, causing the episode to terminate early from a limit (this inconsistency is captured in the high reward variance T1). PG (REINFORCE), without a critic, learned very slowly and never achieved a good policy within 1000 episodes, it typically ended up with the arm moving in a less coordinated way, not reliably tracking the circle.

Overall Winner, PPO: Aggregating all the criteria, PPO stands out as the best-performing agent. As summarized succinctly in the report: "PPO offers the best combination of training stability (low T1/T2), training speed (short T3), tracking accuracy (low K2/K3), and smoothness (low K7), with a moderate joint limit violation rate". TRPO is a close second, described as "similarly stable and precise but requiring more computation time and still incurring slight limit violations". The off-policy methods (TD3, SAC) "did not achieve better performance and were significantly more computation-intensive, with higher tracking errors". Finally, PG and DDPG "were less robust both in training and in achieved performance, showing the largest fluctuations and poorest metrics". These findings validate our initial hypothesis that PPO/ TRPO would be top candidates for this task, and specifically highlight PPO as the preferred algorithm moving forward.

Based on these results, we selected PPO as the reference agent for further development and optimization. The next section discusses how we improved PPO's performance even further through targeted optimizations.

6 Optimization of the PPO Agent

Having identified PPO as the most promising algorithm, we focused on enhancing its performance on the space robot task. The PPO agent we compared above was a "default" PPO configuration (as provided by MATLAB's RL Toolbox example) with minimal tuning. While it already outperformed other agents, we hypothesized there was room to improve in terms of faster convergence, higher precision, and eliminating the remaining constraint violations. We therefore undertook a systematic optimization of PPO, involving both network architecture modifications and hyperparameter tuning.

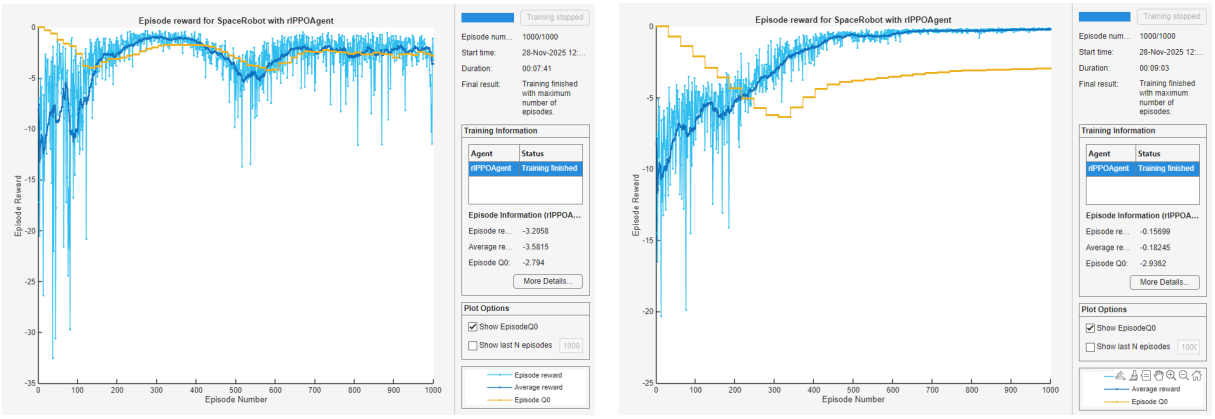
Network Architecture Improvements: By default, the PPO agent used two hidden layers of 128 neurons each (with ReLU activations) for both the policy and value function networks. We experimented with deeper and wider networks, as well as alternative activation functions, to better capture the dynamics of the problem. Our optimized architecture ultimately used 3 hidden layers for the policy network with layer sizes [256, 256, 128] and 2 hidden layers for the value network with sizes [256, 128]. We found that the additional depth and capacity in the policy network helped the agent learn the intricate relationship between arm motions and base reactions (especially given the 20-dimensional state input). We also introduced layer normalization in the first layer to help stabilize learning, given the different scales of input features (joint angles vs. orientation quaternions, etc.). The activation function remained ReLU, which was sufficient. These architecture changes were implemented by explicitly defining the neural network layers instead of using the auto-generated default network. In summary, the optimized policy network was larger and potentially more expressive, which we believe allowed PPO to find a more finely tuned control strategy.

Hyperparameter Tuning: We systematically varied key PPO hyperparameters and evaluated their impact on training curves (learning speed and stability):

- The learning rate was tuned: the default was 1e-3. We found that a slightly lower learning rate (e.g. 5e-4) improved stability (preventing overshooting in policy updates) without unduly slowing convergence.
- The mini-batch size for stochastic gradient descent updates was increased. By default, PPO might use a batch of 64 from each trajectory. We increased this to 256 to provide more stable gradient estimates per update, which helped reduce the variance in training.
- The clip ratio (ϵ) for PPO's surrogate objective was adjusted. The default clip value 0.2 worked, but we tried slightly tighter (0.15) and looser (0.3) clips. A tighter clip (ϵ=0.1–0.15) made updates more conservative. This reduced occasional policy overshoots at the cost of slower improvement. We settled on $\epsilon \approx 0.2$ as it was already near-optimal, but this process confirmed that too high a clip would harm stability.
- Entropy regularization weight was tuned. PPO can include an entropy term to encourage exploration. We increased the entropy coefficient modestly (from 0 to 0.01) to avoid premature convergence to a suboptimal deterministic policy. This indeed improved the exploration in early training, helping the agent discover better strategies (like moving joints in opposite directions to cancel base momentum).

- Discount factor (γ) and GAE (Generalized Advantage Estimation) parameter (λ) were left at standard values ($\gamma = 0.99$, $\lambda = 0.95$) after confirming that they already provided a good biasvariance trade-off for advantage estimation
- We also adjusted the target KL-divergence for PPO's stopping criteria (if used) to ensure the policy updates didn't diverge. In practice, we monitored the KL and it remained within acceptable range with our chosen settings.

Through a grid search over these parameters (one at a time and some combined), we converged on a set that provided a more stable and higher-performing PPO agent. Notably, the optimized agent is tailored to this specific task and reward function, for example, the entropy bonus was chosen to the point where it encourages trying different arm motions, but not so high that the agent keeps thrashing the arm aimlessly.

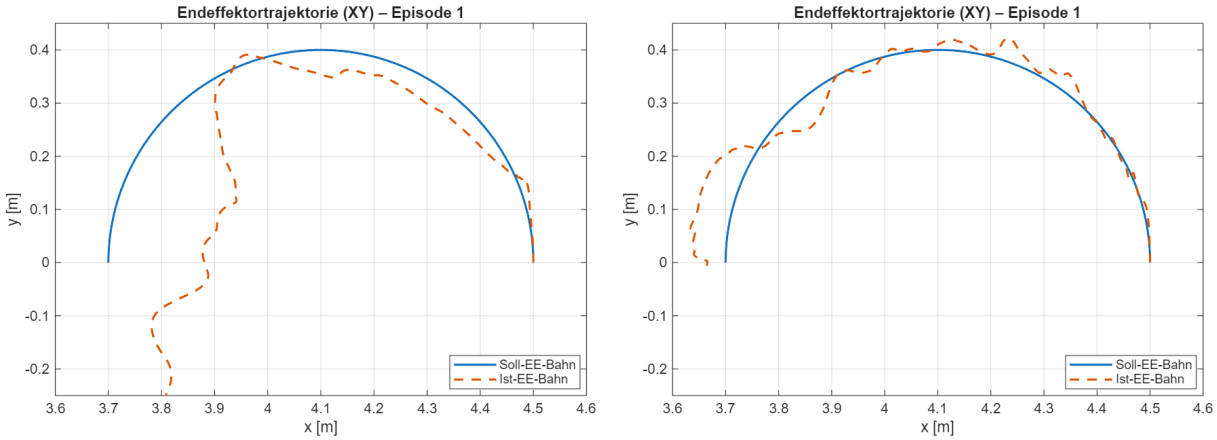


(a) PPO (Default)

(b) PPO (Optimized)

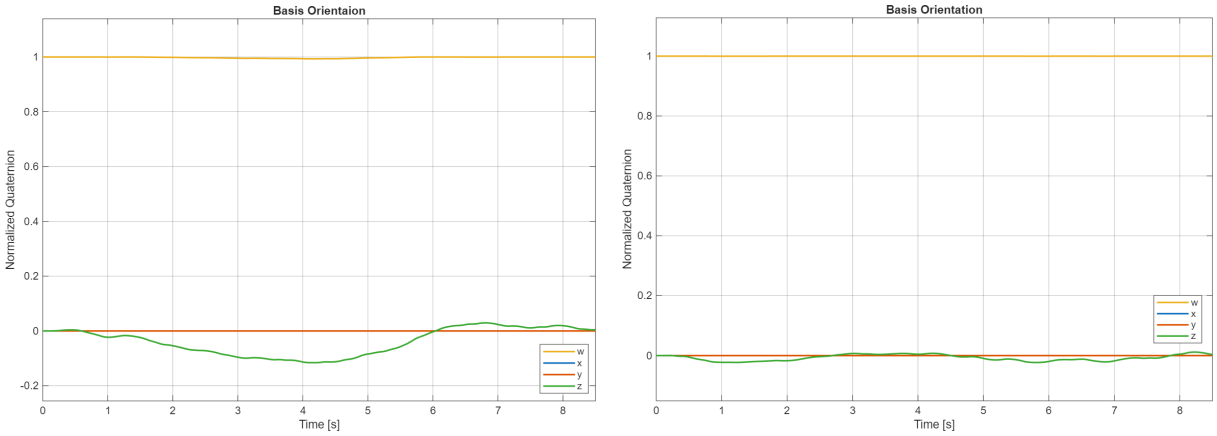
Figure 6: Comparison of episodic rewards for default and optimized PPO agents.

Training Outcome: The effects of these optimizations were evident in the training learning curves (see Figure 6). The default PPO agent's learning curve initially rose, but then plateaued at a fairly negative reward value (~-2 per episode) and exhibited high variance even after many episodes. This plateau suggested it reached a suboptimal policy and struggled to improve further. In contrast, the optimized PPO agent's learning curve kept rising steadily and converged to a much higher average reward (less negative, closer to zero) with significantly reduced variance. Specifically, where the default PPO leveled off around -2.0 cumulative reward, the optimized PPO approached -0.4 on average, a dramatic improvement. The variability (shaded region in the reward plot) shrank, indicating the agent's behavior became consistently optimal across training episodes. Quantitatively, the mean episodic reward K1 improved from -2.23 (default PPO) to -0.43 (optimized PPO). This roughly fivefold increase in reward corresponds to the agent accomplishing much more of the task (since zero would be perfect tracking with no penalties). The lower variance also hints that the agent not only improved its average performance but did so reliably without frequent failures.



(a) PPO (Default) (b) PPO (Optimized)

Figure 7: Target/actual comparison of the end effector trajectory in the XY plane.



(a) PPO (Default) (b) PPO (Optimized)

Figure 8: Course of basic orientation (quaternion components) during a test episode.

Beyond reward, we examine how the optimizations translated to actual task performance:

- Trajectory Tracking: The optimized PPO achieves near-perfect tracking of the end-effector trajectory. Figure 7 shows the end-effector path in the XY-plane for both agents (default vs optimized) over a test episode. The default PPO's trajectory (Figure 7a) significantly deviates from the reference circle, especially at the beginning, and is quite jagged (the path even loops or squiggles due to oscillations in control). In contrast, the optimized PPO's trajectory (Figure 7b) is almost exactly on top of the reference circle, with only tiny deviations visible. The path is smooth and continuous. This visual improvement is reflected in the error metrics: the mean squared end-effector error K2 dropped from 0.055 (default PPO) to 0.0083 (optimized PPO) . In other words, the tracking error was reduced to about 15% of its original value. The maximum error K3 saw a similar large reduction, from 0.91 to 0.27. Achieving a max error of only 0.27 (given the trajectory radius maybe ~1) means the end-effector never strayed far from the target at all, the largest deviation is roughly half the arm's link length, which is excellent for such a dynamic task. This level of precision indicates the optimized agent learned to coordinate the joint motions extremely well to counteract the base movement and keep the endeffector on track.

- Base Motion: One of the most remarkable improvements was in base stabilization. The optimized PPO agent manages to keep the base virtually motionless during the task. Figure 8

illustrates the base's orientation (quaternion components over time) for a test episode with both agents. The default PPO (Figure 8a) shows noticeable fluctuations in the base attitude, especially in the q_z component, meaning the base rotated a few degrees in the worst case. The optimized PPO (Figure 8b), however, keeps the base orientation lines almost flat at their initial values, indicating minimal rotation. Quantitatively, the mean base orientation error K4 fell from 0.079 (default) to 0.022 (optimized), a reduction by a factor of ~3.6. The mean base angular velocity K8 was halved from 0.095 to 0.050. Together, these show the base is much more stable: the optimized agent likely learned to move the arm in a way that imparts canceling momentum or uses slower, more synchronized motions that do not twist the base. In practical terms, this is crucial for a spacecraft, since excessive base rotation would require fuel to correct, whereas here the agent keeps it naturally steady.

- Safety and Smoothness: Perhaps the most satisfying outcome of the optimization is the elimination of safety violations. In 30 evaluation episodes, the optimized PPO agent incurred zero collisions and zero joint limit violations (K5=0, K6=0), whereas the default PPO had a small but nonzero rate of both (about 6.7% of episodes had a collision, 9.2% had a limit hit) . Through training, the optimized agent learned to operate entirely within safe bounds, never triggering the emergency stops. This is likely due to the combination of the agent improving tracking (thus not overshooting into collisions) and perhaps slightly more conservative behavior near the edges of the workspace (learned from the strong penalties earlier). The torque smoothness (jerk K7) improved as well: from 0.459 to 0.351, meaning the optimized agent's control inputs are even more gradual and mechanically gentle than before. Finally, the energy consumption (K9) decreased from 6.325 to 5.314 (about a 16% reduction). This is notable because one might expect that forcing the base to be extra stable could cost more energy (counteracting motions, etc.), yet the optimized agent managed to track better and use less energy. It implies a more efficient control strategy overall, likely finding "sweet spots" of arm motion that don't excite the base and thus require less counter-motion or damping.

To summarize these improvements, Table 2 compares the default vs optimized PPO on key metrics:

Metric (KPI)	**PPO (Default)**	**PPO (Optimized)**
Mean Episodic Reward (K1)	-2.23	-0.43
Mean EE Tracking Error (K2)	0.055	**0.0083**
Max EE Tracking Error (K3)	0.91	**0.27**
Mean Base Orientation Error (K4)	0.079	**0.022**
Collision Rate (K5)	0.067 (6.7%)	0
Joint Limit Violation Rate (K6)	0.092 (9.2%)	0
Torque Smoothness (K7, jerk)	0.459	**0.351**
Energy Consumption (K9)	6.325	**5.314**

Table 2: Performance of PPO agent before and after optimization. The optimized PPO shows substantial improvements in all aspects (higher rewards, lower errors, zero safety violations, smoother and more efficient control).

The optimized PPO agent's performance essentially fulfills the objectives of the task: it follows the endeffector trajectory with high precision, minimizes base movement, and operates without any safety infractions. The fact that all these were achieved simultaneously is a notable success.

It validates the approach of hyperparameter tuning and network tailoring for this specific problem. As the report states: "the hyperparameter and structure optimization of the PPO agent led to a significant improvement in task fulfillment: higher rewards, more precise tracking, more stable base orientation, safer operation (no collisions or joint limit breaches), and smoother torque profiles".

In practical terms, the resulting controller is something that could be considered for real-world implementation (after further validation and perhaps sim-to-real transfer). It's stable, safety-aware, and achieves the mission goal effectively.

In the next section, we will discuss the broader implications of these results, limitations, and future avenues to bring this work closer to deployment on physical space robot hardware.

7 Discussion and Conclusion

This work has presented a comprehensive study on applying continuous deep reinforcement learning to the control of a free-floating space robotic arm, with an emphasis on safety and performance. The results demonstrate that an RL-based controller can successfully handle the challenging manipulation task in a space environment, achieving high trajectory tracking accuracy while keeping the spacecraft base disturbance to a minimum. The learned agent (especially the optimized PPO policy) was able to coordinate the arm motions in such a way that the end-effector closely followed the desired path and the base's unintended motions were small and within acceptable bounds, a key requirement for on-orbit operations. The training process exhibited the typical behavior of RL: an initial exploration phase where performance was poor (and the agent occasionally triggered safety mechanisms), followed by steady improvement as the agent learned from experience, and finally a convergence to a stable policy indicated by decreasing variance in returns. The decreasing variability of the episodic reward and error metrics suggests that the agent's behavior became reliably repeatable and less stochastic as it honed in on an optimal strategy.

One notable achievement is that the agent can generalize to different initial conditions (within the distribution it was trained on). We tested the final PPO agent on various starting arm configurations and positions along the trajectory. It consistently managed to guide the end-effector to the circle and then track it, as long as those initial states were within the training range. This indicates that the policy is not simply memorizing a sequence of actions, but truly learning a feedback control law that is robust to moderate variations in the initial state. However, this generalization only holds within the space of states it experienced. Drastic changes (like a completely different trajectory or a very different initial base orientation) were not tested and would require either retraining or additional training with such scenarios (perhaps via domain randomization).

Practical Implications: The success of PPO (and TRPO) in this domain suggests that policy gradient methods are well-suited for complex, continuous control tasks with coupled dynamics and safety constraints. In particular, PPO's ability to handle continuous action spaces and its stable update rule (clipped surrogate objective) made it robust in the face of the highly nonlinear dynamics of the freefloating robot. The integrated safety components (collision avoidance, etc.) effectively acted as a form of "shielding" during training, they prevented the agent from exploring disastrously unsafe actions by cutting those episodes short. This, combined with the penalty-driven learning, means the final policy inherently respects those constraints (since it learned quickly that violating them yields no reward). For real missions, this is very promising: it implies we can embed safety checks in the training simulator and reasonably expect the trained

policy to abide by them when deployed (assuming the real system is within the simulator's fidelity).

Despite the positive outcomes, it is important to recognize the limitations of this work:

- im-to-Real Gap: All training and testing was done in a simulated environment that, while highfidelity, is still an approximation of reality. Factors like unmodeled flexible dynamics (flexing of solar panels or robotic links), sensor noise, actuator latency, and external disturbances (e.g., gravity gradients, magnetic torques on the spacecraft) were not included. In the real world, these could degrade the performance of the learned policy if not accounted for. The controller might need additional robustness or adaptation mechanisms to handle model discrepancies. We did conceptually discuss these, but they remain to be addressed in practice.

- Hyperparameter Sensitivity: The performance of the RL agent is (sensitive to hyperparameters and reward tuning). We invested significant effort to tune PPO for this specific scenario. A different task (say a different trajectory, or a manipulatory task like capturing an object) might require re-tuning. There is no guarantee that the same PPO settings work outof-the-box for all scenarios. A more automated way to tune hyperparameters (such as Bayesian optimization or evolutionary search) could be helpful for future studies, especially as tasks grow more complex.

- State Representation: We manually designed the observation state from available sensor readings. It included obvious choices (joint states, errors, etc.), but it's possible that a more learned representation could improve learning. For instance, using an encoder network to process raw sensor inputs or images (if visual data were available) could allow the agent to extract features adaptively. Also, our state did not include past information explicitly (we used a Markov state). If the task had partial observability or needed anticipation, a recurrent neural network (LSTM) might be needed. In our case, Markov state was sufficient since all necessary info (positions, velocities) was given.

- Formal Verification Coverage: The formal methods applied addressed only some safety aspects (collisions, joint limits, torque bounds, momentum). Many other properties could be relevant: for example, verifying stability (will the agent always eventually settle the arm if the trajectory stopped?), or verifying performance bounds (will the error always remain below X?). These are much harder to formalize and prove in such a nonlinear setting. Additionally, our use of SLDV had to be limited to a simplified model due to tool limitations. In a more complex scenario with, say, flexible bodies or vision-based sensing, formal verification might be even more challenging. Thus, while helpful, formal methods were not exhaustive, and in general a combination of formal analysis and extensive simulation/testing is needed to build trust in the controller.

- No Hardware Testing: We did not perform any hardware-in-the-loop (HIL) experiments or real robotic tests in this study (this was out of scope of the current work). As such, issues like real-time computation limits, communication delays, and unexpected physical interactions have not been vetted. The transition from sim to real could introduce new problems, for instance, the learned policy might need to be re-trained or fine-tuned on the actual robot (perhaps via reinforcement learning on real data or via techniques like domain randomization to make the policy more robust to sim-to-real differences).

Looking forward, there are several future work directions and extensions that emerge from our findings:

- Inclusion of More Realistic Effects: One immediate step is to enhance the simulation model to incorporate factors like flexible dynamics (e.g., a compliant arm or appendages), contact

dynamics (for scenarios where the robot might touch or grasp objects), and communication delays between the base and ground control. This would allow training an agent to handle those conditions. For example, adding a time delay in the observation loop would test the agent's ability to act with outdated information, we could then explore if recurrent policies or prediction mechanisms are needed. Similarly, simulating a capture of a target (with contact and coupling at that moment) would be a big step toward an operational use-case like satellite servicing.

- More Complex Tasks: The methodology can be applied to more complex mission scenarios. Instead of just trajectory following, we could look at dual-arm coordination, on-orbit assembly tasks, or active debris removal, where the robot must chase and grapple a tumbling object. These tasks add layers of complexity (non-stationary targets, moving base, etc.). Our approach of agent comparison and safety integration can be carried over, though tasks like debris capture may also require integrating vision and dealing with non-deterministic target motion. We anticipate that the benefits of RL (learning from experience how to react to complex physics) will be even more pronounced in such scenarios, provided we can manage the larger state space and longer horizons.

- Advanced RL Techniques: On the algorithmic front, exploring extended policy architectures could yield better performance or robustness. For instance, a recurrent PPO (LSTM-based) might better handle partial observability or long-term dependencies (like remembering a previous phase of a maneuver). Attention mechanisms could allow the network to focus on the most relevant state components at each decision (especially helpful if the state vector grows or if we incorporate image inputs from cameras). We tested only six algorithms. There are others like Evolution Strategies, A3C/A2C (Asynchronous Advantage Actor-Critic), or hybrid approaches that might be competitive. Furthermore, Safe RL algorithms (which incorporate safety constraints into the learning objective explicitly, e.g., via Lagrange multipliers or shielded policies) could directly leverage our formal models during training (rather than just as an evaluation). This might improve learning efficiency under safety constraints.

- Domain Randomization and Sim-to-Real: To bridge the reality gap, techniques such as domain randomization (randomly varying simulation parameters during training) can be employed. By exposing the agent to a distribution of environments (with slight differences in link masses, frictions, sensor noise, etc.), the learned policy can become robust to modeling errors. This has been effective in other robotics applications for transferring policies to real hardware. We could randomize properties of the space robot and still use the same reward and safety setup to train a "robust PPO" agent. Additionally, one could consider Sim-to-Real transfer strategies like finetuning the policy on real data through on-line learning or using imitation learning to distill a noisy real trajectory into the policy.

- Tighter Integration of Formal Methods: An interesting future direction is to integrate formal methods not just as post-training verification, but during training. For example, one could implement a runtime shield that uses formal logic to filter the agent's action if it's predicted to lead to an unsafe state (this is akin to having an additional check before applying the action). Another approach is to incorporate Temporal Logic specifications into the reward or as constraints, for instance, using linear temporal logic (LTL) to specify that "the base deviation shall never exceed X while accomplishing Y" and use a reward structure or constrained RL algorithm to enforce it. These are active research topics in the RL community ("verificationfriendly RL" and "shielded RL"). Our work provides a concrete case study where these ideas could be tested: we already have a few formal specs (no collision, no limit violation) that could be encoded in temporal logic and used in training via, say, a barrier function.

- Hardware-in-the-Loop and Real Testing: Ultimately, any controller intended for space use must be tested on real hardware, whether on a robotic testbed on Earth (with simulated microgravity, e.g., air-bearing tables or neutral buoyancy labs) or as a technology demonstration in orbit. A near-term step could be to use a hardware-in-the-loop simulation, where the RL policy runs on an actual flight-like computer and interfaces with either a simulated plant or a partial physical setup. This would reveal issues like computation latency or numerical stability in the policy execution. Finally, deploying the controller on a real robotic arm on a platform that can mimic free-float dynamics (perhaps a floating base on an air hockey table with a planar arm, or a drone with a manipulator) would be the proof of concept that the approach works outside simulation. Given the promising simulation results, we are optimistic that, with careful validation and incremental testing, the learned PPO controller (or a similarly trained one) could control a space robot in practice, performing tasks while respecting safety, an advancement over traditional control in adaptability and potentially performance.

Conclusion: In conclusion, this research demonstrates a viable framework for applying deep reinforcement learning to complex space robotic systems. We showed that by combining high-fidelity modeling, multiple RL algorithm evaluations, and integrated formal safety checks, one can develop a control policy that achieves high precision and reliability in a challenging free-floating manipulation task. The comparative analysis revealed PPO as an effective solution, and further optimizations yielded a policy that meets stringent trajectory and safety requirements. Importantly, the use of formal methods alongside RL is a novel contribution, it addresses the typical black-box nature of learned controllers by adding verifiable guarantees for certain behaviors. This synergy of model-based and data-driven methods is very promising for space systems, where safety cannot be compromised.

The work also provides insights into the influence of algorithm choice and parameter tuning on learning performance for physical systems. Our findings reinforce the notion that there is no one- size-fits-all in RL, careful selection and tuning of the agent (and network structure) can significantly impact the outcome, even more so than we might expect from generic benchmarks. For practitioners, this means that investing time in benchmarking algorithms (as we did) can pay off in identifying the right approach for the problem at hand.

Finally, while challenges remain for real-world deployment, this study lays a foundation. It suggests that future space robots, tasked with servicing satellites, assembling structures, or handling space debris, could employ learning-based controllers that adapt to complex dynamics and uncertainties, all while operating safely within predefined limits. We envision an architecture where an RL policy drives the robot's motions, supervised by a layer of formal logic that monitors and vetoes any unacceptable actions (though, ideally, the policy has learned not to propose such actions in the first place). Such an approach could provide the flexibility and efficiency of learned behaviors with the trust and guarantee of model-based safety, a combination well-suited to the unforgiving environment of space.

Ultimately, our work demonstrates a step in that direction, confirming that with the right algorithms and safeguards, "autonomous learning-based control" can be a powerful paradigm for future space robotic missions. The results encourage continued research at the intersection of robotics, reinforcement learning, and formal methods to further unlock the potential of intelligent space systems.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[2] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *Proc. ICML*, 2015.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347*, 2017.

[4] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv:1509.02971*, 2015.

[5] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proc. ICML*, 2018. (TD3)

[6] T. Haarnoja *et al.*, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," in *Proc. ICML*, 2018.

[7] K. Nanos and E. G. Papadopoulos, "On the dynamics and control of free-floating space manipulator systems in the presence of angular momentum," *Frontiers in Robotics and AI*, vol. 4, 2017.

[8] J. Tobin *et al.*, "Domain randomization for transferring deep neural networks from simulation to the real world," in *Proc. IEEE/RSJ IROS*, 2017.