Demonstration of the Dyna Reinforcement Learning Framework for Reactive Close Proximity Operations

Ritwik Majumdar[*]
University of Michigan, Ann Arbor, Michigan, USA

David C. Sternberg[†], Keenan Albee[‡]
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, USA

Oliver Jia-Richards[§]
University of Michigan, Ann Arbor, Michigan, USA

Lessons from the International Space Station (ISS) emphasize the necessity of exterior inspection for anomaly detection and maintenance, but current methods rely on costly and limited human extravehicular activities and robotic arms. Deployable free-flying small spacecraft offer a flexible, autonomous solution, capable of comprehensive exterior inspections without human involvement. However, the safety of these spacecraft during close proximity operations remains a concern, particularly given uncertain variability in thruster performance. This paper presents SmallSat Steward, a reactive and integrated architecture for online model learning and trajectory planning based on the Dyna reinforcement learning architecture. By combining model-based planning and direct reinforcement learning, Dyna offers a potentially flexible and computationally efficient solution capable of adapting to changes in thruster performance and other system uncertainties. Preliminary results in both simulation and hardware environments demonstrate the potential of this architecture to successfully regulate position under single and double thruster failures. In simulation, the Dyna-based controller outperformed a PD-LQR controller in ~70% of all cases. On hardware, Dyna was able to eliminate the steady state error caused by thruster failures.

I. Introduction

Lessons learned from the International Space Station (ISS) in low-Earth orbit have demonstrated the value of exterior inspection for detecting anomalies or assessing maintenance needs. However, current inspection capabilities on the ISS rely on costly human extra-vehicular activities or imagery taken by robotic arms mounted to the exterior of the station and visiting vehicles (e.g. cargo resupply missions). These approaches detract from valuable crew time or severely limit the possible visual coverage of the station's exterior—mounted robotic arms cannot survey the entire exterior of the station and visiting vehicles may be infrequent or need to operate within high safety margins.

Deployable free-flying small spacecraft can provide a unique and cost-effective solution for exterior inspection of future space stations that would resolve the limitations present in current inspection techniques. These spacecraft can be stowed, deployed as needed, and re-stowed in order to autonomously perform regular or on-demand exterior inspection. Furthermore, these spacecraft might also perform light extra-vehicular servicing work which currently requires extensive crew effort. Such an approach could provide multiple benefits over traditional methods for exterior inspection: no human involvement is required; a free-flying small spacecraft can access the entire exterior surface rather than the limited regions accessible to a mounted robotic arm; and the small spacecraft can be deployed on demand rather than waiting for a visiting vehicle.

However, safety is key for any free-flying spacecraft during close proximity operations, primarily in terms of avoiding collisions between the free-flying spacecraft and the station. Passive CubeSats that are ejected away from the station [1]

[*]Ph.D. Student, Department of Aerospace Engineering, ritwikm@umich.edu, AIAA Student Member.

[†]Guidance and Control Systems Engineer, Payload/Instrument Pointing Control Systems Engineering Group, david.c.sternberg@jpl.nasa.gov, AIAA Senior Member.

[‡]Robotics Technologist, Maritime and Multi-Agent Autonomy Group, keenan.albee@jpl.nasa.gov.

[§]Assistant Professor, Department of Aerospace Engineering, Department of Climate and Space Sciences and Engineering, oliverjr@umich.edu, AIAA Member.

alleviate safety considerations but suffer from similar visibility constraints to mounted robotic arms and are single-use. Actively-controlled CubeSats [2, 3] enable full coverage of the exterior of the station, but can collide with the station in the event of common small satellite failures [4]. Current considerations for avoiding collisions—other than extreme conservatism not possible in close proximity operations—rely on the ability to use onboard thruster systems to perform collision-avoidance maneuvers as well as maintaining accurate knowledge about the thruster system's performance [5, 6]. However, thruster performance can change over time, with the recent Seeker mission predicting changes in the thrust output of up to 70% during the brief demonstration mission [7]. These performance changes can be difficult to predict, and may result in the inability to accomplish maneuvers designed under an assumed thruster performance level. Such was the case with NASA/JPL's Lunar Flashlight mission, which experienced a wide range of unexpected thrust levels that eventually prevented it from reaching the intended lunar orbit [8].

Accounting for thruster performance changes and, more broadly, system model uncertainty, can be a computationally-demanding task. Recent algorithm developments that rely on nonlinear trajectory optimization to plan trajectories that resolve model uncertainties [9] might be beyond the real-time capability of traditional small spacecraft computers while only targeting parametric uncertainties. Compute constraints are especially of interest for small (3–6U) CubeSats, which would be the desired form factor for an external free-flying inspector spacecraft, providing a balance between a compact form factor and sufficient payload volume and onboard capability to perform an inspection mission. This computational boundary and modeling flexibility represents the technology gap our work aims to resolve: no known planning solution in the literature is amenable to implementation on small spacecraft computers while providing capability for online planning and learning with varied disturbance sources. This gap prevents small spacecraft from reliably performing close proximity operations tasks like autonomous inspection, as the spacecraft would not be able to safely plan and fulfill inspection objectives under significant model degradation.

The broad context of this work is to introduce SmallSat Steward, a reactive and integrated architecture for online model learning and trajectory planning based on the Dyna reinforcement learning architecture [10]. Dyna combines both planning and learning: it is both a model-based and direct reinforcement learning framework, where real experience is used to provide direct updates of a learned value function and to update a system model that is available for model-based planning updates of the same value function. It is anytime in the sense that a solution can always be constructed from a prior initialization of the saved value function and real-time in the sense that updates to the value function are computationally cheap, and simulated planning updates are allowed to execute within a specified time bound. Dyna is appealing in comparison to numerical trajectory optimization methods since solutions are guaranteed, runtime can be hard bounded, and the use of linear model updates is computationally simple. It is also appealing in comparison to alternative reinforcement learning methods as the use of a learned model for simulated value function updates allows the agent to avoid failure states without having to experience them.

This paper presents preliminary results for the application of Dyna towards reactive proximity operations in both simulation and hardware environments. The demonstration of interest is to show the Dyna architecture's capability to respond to onboard thruster degradation or failure for a simple regulation scenario, where the inspector spacecraft is tasked with maintaining a fixed position relative to the target space station. A variety of failure causes were tested with thruster failure modes ranging from partial degradation to full failure. The demonstration was conducted both in a simulation environment and on a hardware testbed, using a thrusting planar air bearing platform [11], to demonstrate the implementation of Dyna on flight-like CubeSat processors.

II. Problem Formulation and Description

Figure 1 shows an example of a small, free-flying spacecraft deployed to inspect the exterior of a space station. The inspection point has been predetermined, and the spacecraft is tasked with navigating to it autonomously, performing a task (e.g., taking a picture), and returning to the space station for recovery. The spacecraft has been provided with waypoints that avoid the station and provide a safe path to the inspection point. The thruster array onboard the spacecraft has a low probability of degrading or failing unexpectedly, and the onboard controller must be able to handle these failures to protect the spacecraft and station as well as complete the mission objective.

A. Dynamics

The scenario detailed above can be modified to be a sequence of position regulation tasks. For each regulation task, we attempt to drive the spacecraft to a given reference state $\boldsymbol{s}_{\text{ref}}$ (the waypoint). We assume that our spacecraft's state

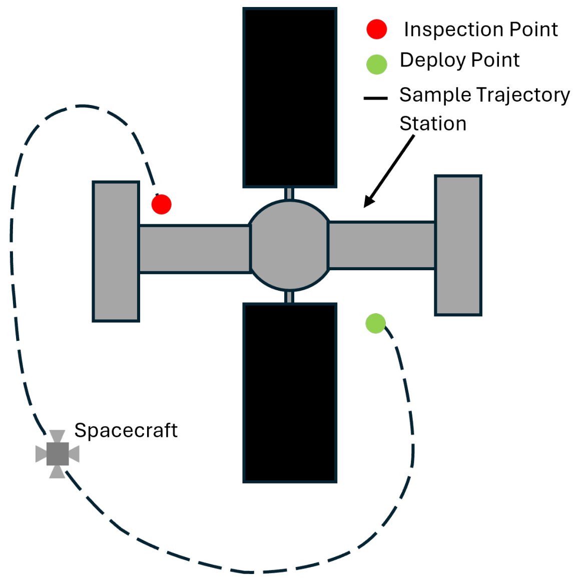

Fig. 1 An example inspection task.

vector, $\boldsymbol{s}_{\text{s/c}}$, contains two positional degrees of freedom and one rotational degree of freedom

$$\boldsymbol{s}_{\text{s/c}} = \begin{bmatrix} x & y & \theta & v_x & v_y & \omega \end{bmatrix}^T \tag{1}$$

which mimics the degrees of freedom for the planar air bearing hardware testbed, described in Section V.A. For a position regulation task, we attempt to drive the relative state, $\boldsymbol{s}$, to the zero vector, $\mathbf{0}$.

$$\boldsymbol{s} = \boldsymbol{s}_{\text{s/c}} - \boldsymbol{s}_{\text{ref}} \tag{2}$$

The controller will output high-level actions in the form of the desired net forces and torques in the reference frame

$$\boldsymbol{a} = \begin{bmatrix} F_x & F_y & \tau \end{bmatrix}^T \tag{3}$$

The dynamics of the spacecraft's state relative to the reference point are given by the standard Newton-Euler equations for two positional degrees of freedom and one rotational degree of freedom in discrete time. Given the spacecraft's relative state, $\boldsymbol{s}$, at some time t, the spacecraft's updated relative state, $\boldsymbol{s}'$, at some time $t + \Delta t$ is

$$\boldsymbol{s}' = [A]\boldsymbol{s} + [B]\boldsymbol{a} \tag{4}$$

where $[A]$ and $[B]$ are defined as

$$[A] = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad [B] = \begin{bmatrix} \Delta t^2/2m & 0 & 0 \\ 0 & \Delta t^2/2m & 0 \\ 0 & 0 & \Delta t^2/2I \\ \Delta t/m & 0 & 0 \\ 0 & \Delta t/m & 0 \\ 0 & 0 & \Delta t/I \end{bmatrix} \tag{5}$$

and m is the spacecraft's mass while I is the spacecraft's rotational inertia about its single rotational degree of freedom. For all testing, $\Delta t = 0.2$ s, $m = 14.5$ kg, and $I = 0.35$ kgm^2 to mimic a 5 Hz control loop using the planar air bearing hardware testbed, described in Section V.A. The choice to mimic the setup of the hardware (including its dynamics) was made out of practicality. The simpler dynamics allow us to compare to known optimal control algorithms. Additionally, matching hardware dynamics allows for better comparison between hardware and simulated testing and informed design of the architecture (such as neural network sizing). However, the overall design of the actual implementation will be able to handle nonlinear and more complex dynamics.

B. Constraints

Throughout all testing we define a "safe" zone, $\{\chi_{\text{safe}}\}$, around the reference point that constrains the allowable relative state of the spacecraft. This constraint helps with training times, but is realistic in actual mission scenarios, as mission trajectories usually have defined "keep-in" zones to offer some margin of safety. [12] For the purposes of this problem, we define $\{\chi_{\text{safe}}\}$ as

$$\{\chi_{\text{safe}}\} = \{\boldsymbol{s} \mid |\boldsymbol{s}| < \boldsymbol{c}_{\text{safe}}\} \tag{6}$$

where

$$\boldsymbol{c}_{\text{safe}} = \begin{bmatrix} 5 \text{ m} & 5 \text{ m} & \pi & \infty & \infty & \infty \end{bmatrix}^T \tag{7}$$

which constrains the spacecraft position to lie in a 5 m square around the reference point, and the spacecraft's rotation to lie on the range $\theta \in [-\pi, \pi]$. Similar to the state constraints, we also define action constraints arising from hardware limitations. We define a set of feasible actions, $\{\chi_{\text{action}}\}$, of all possible valid actions as

$$\{\chi_{\text{action}}\} = \{\boldsymbol{a} \mid |\boldsymbol{a}| < \boldsymbol{c}_{\text{action}}\} \tag{8}$$

where

$$\boldsymbol{c}_{\text{action}} = \begin{bmatrix} 1.0245 \text{ N} & 1.0245 \text{ N} & 0.1682 \text{ Nm} \end{bmatrix}^T \tag{9}$$

which represents the capabilities of the thruster system onboard the planar air bearing hardware testbed.

Finally, we also define a set of goal states, $\{\chi_{\text{goal}}\}$ as

$$\{\chi_{\text{goal}}\} = \{\boldsymbol{s} \mid |\boldsymbol{s}| < \boldsymbol{c}_{\text{goal}}\} \tag{10}$$

where

$$\boldsymbol{c}_{\text{goal}} = \begin{bmatrix} 1 \text{ mm} & 1 \text{ mm} & 1 \text{ mrad} & 1 \text{ mm/s} & 1 \text{ mm/s} & 1 \text{ mrad/s} \end{bmatrix}^T \tag{11}$$

During training, $\{\chi_{\text{goal}}\}$ is used as a terminal state set to account for the fact that the agent will never be exactly at the desired reference point.

C. Markov Decision Process Formulation

To determine the appropriate control inputs for the position regulation task, we formulate the regulation problem as a Markov decision process (MDP). In an MDP, an agent operates with a policy that determines the action to take when the spacecraft is in a given state, $\boldsymbol{a} \leftarrow \pi(\boldsymbol{s})$. The goal of the agent is to determine the optimal policy, π^*, that maximizes the discounted expected cumulative reward

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R(\boldsymbol{s}_i, \boldsymbol{a}_i)\right] \tag{12}$$

where γ is the discount factor, i is the time index, and R is the instantaneous reward function. The instantaneous reward is a function of both the spacecraft state and the chosen control action. In this work, the reward function is defined as

$$R(\boldsymbol{s}, \boldsymbol{a}) = -(\boldsymbol{s}^T [Q] \boldsymbol{s} + \boldsymbol{a}^T [R] \boldsymbol{a}) \Delta t + r_{\text{safe}}(\boldsymbol{s}) + r_{\text{goal}}(\boldsymbol{s}) \tag{13}$$

where the matrices $[Q]$ and $[R]$ are defined according to

$$[Q] = \operatorname{diag}\left(\begin{bmatrix} 0.2 & 0.2 & 1/\pi & 0 & 0 & 0 \end{bmatrix}\right) \qquad [R] = \operatorname{diag}\left(\begin{bmatrix} 0.9761 & 0.9593 & 5.9436 \end{bmatrix}\right) \tag{14}$$

and the functions r_{safe} and r_{goal} augment the reward according to whether the spacecraft is in the safe zone and/or goal region

$$r_{\text{safe}}(\boldsymbol{s}) = \begin{cases} 0 & \boldsymbol{s} \in \{\chi_{\text{safe}}\} \\ -100 & \boldsymbol{s} \notin \{\chi_{\text{safe}}\} \end{cases} \qquad r_{\text{goal}}(\boldsymbol{s}) = \begin{cases} 100 & \boldsymbol{s} \in \{\chi_{\text{goal}}\} \\ 0 & \boldsymbol{s} \notin \{\chi_{\text{goal}}\} \end{cases} \tag{15}$$

The values for the diagonal elements of the $[Q]$ and $[R]$ matrices were selected in this work to be equal to the element-by-element inverse of the state and action bounds in Eqs. 7 and 9.

The reward function is inspired by a traditional linear-quadratic regulator (LQR), where the equivalent policy is defined as a linear feedback controller

$$\boldsymbol{a} = -[K]\boldsymbol{s} \tag{16}$$

where the gain matrix, $[K]$, is calculated from

$$[K] = ([R] + [B]^T[P][B])^{-1}[B]^T[P][A] \tag{17}$$

and $[P]$ is the solution to the infinite-horizon discrete algebraic Riccati equation

$$[P] = [A]^T[P][A] - ([A]^T[P][B])([R] + [B]^T[P][B])^{-1}([B]^T[P][A]) + [Q] \tag{18}$$

In the absence of state and action constraints as well as the additional rewards for being in the safe zone and in the goal region, the LQR policy would be the optimal policy. The goal of the reinforcement learning agent is to understand how to modify the policy due to the state and action constraints as well as the potential for propulsion system degradation and failure. Throughout this work, the traditional LQR controller will be used as a reference policy in order to understand the relative performance of the reinforcement learning agent.

III. Reinforcement Learning Implementation

Dyna is a reinforcement learning architecture that combines learning and planning [10]. In Dyna, real-world experiences (i.e., observed state transitions) are used to update an internal state transition model. Between actions, the agent uses the learned state transition model to simulate experiences that can be used to further update the policy. In effect, Dyna augments traditional reinforcement learning algorithms by enabling the agent to plan. The key benefits of such an architecture for this work are that the agent will be able to re-plan its policy in the event of propulsion system degradation or failure and that the agent will be able to explore and test actions in simulation without having to actually experience failure states. These benefits are why Dyna is being applied to solve this problem, the overall architecture is simple and flexible enough to run on flight-like hardware while containing the necessary components to adapt in real-time to unexpected model changes.

Algorithm 1 shows the implementation of the Dyna architecture used in this work. The outer control loop is similar to any standard reinforcement learning algorithm: the agent observes its current state, takes an action according to its policy, and uses the observed new state and reward to update its state-action value approximation. The incorporation of Dyna adds an inner planning loop based on a learned state transition model. The planning loop simulates trajectories from the current spacecraft's state using the learned state transition model and performs value function updates using the simulated results. The sampling approach used for Dyna planning is further described in Section III.C.

A. State Representation

The state representation, $\boldsymbol{s}$, uses an angular representation for the attitude. The values for θ are restricted to range between $-\pi$ and π from a given reference angle, creating a discontinuity as the spacecraft's attitude crosses from π to $-\pi$ or vice versa. To rectify this, we adopt an extended state representation, $\boldsymbol{s}_{ext}$, which uses the cosine and sine of the angular position as the attitude representation

$$\boldsymbol{s}_{\text{ext}} = \begin{bmatrix} x & y & \cos\theta & \sin\theta & v_x & v_y & \omega \end{bmatrix}^T \tag{19}$$

This extended representation is commonly used inside neural networks to improve the accuracy and performance of the agent [13].

B. Twin-Delayed Deep Deterministic Policy Gradient (TD3)

In this work, our chosen reinforcement learning algorithm is the twin-delayed deep deterministic (TD3) policy gradient algorithm. TD3 was used as the reinforcement learning algorithm as it is an off-policy algorithm which makes

Algorithm 1 Dyna Algorithm with Trajectory Sampling

```
1: Initialize state-action value functions $Q_{1/2}(\boldsymbol{s}_{\text{ext}}, \boldsymbol{a})$, policy $\pi(\boldsymbol{s}_{\text{ext}})$, and model $M(\boldsymbol{s}_{\text{ext}}, \boldsymbol{a})$
2: while not done do
3:     Get current agent state, $\boldsymbol{s}_{\text{ext}}$
4:     Get action, $\boldsymbol{a} \leftarrow \pi(\boldsymbol{s}_{\text{ext}})$
5:     Take action $\boldsymbol{a}$, observe new state $\boldsymbol{s}_{\text{ext}}{}'$, and calculate reward $R(\boldsymbol{s}', \boldsymbol{a})$
6:     Update $Q_{1/2}(\boldsymbol{s}_{\text{ext}}, \boldsymbol{a})$ and $\pi(\boldsymbol{s}_{\text{ext}})$
7:     Update $M(\boldsymbol{s}_{\text{ext}}, \boldsymbol{a})$ based on observed state transition
8:     Update $\alpha$ based on transition loss $loss_t$
9:     for $n$ trajectories do
10:        Initialize simulation state with current agent state $\boldsymbol{s}_{\text{ext,sim}} \leftarrow \boldsymbol{s}'_{\text{ext}}$
11:        Create empty trajectory $T(\boldsymbol{s}_{\text{ext}}, \boldsymbol{a}, \boldsymbol{s}_{\text{ext}}{}', \boldsymbol{r})$
12:        for $l$ look-ahead steps do
13:            Get simulated action $\boldsymbol{a}_{\text{sim}} \leftarrow \epsilon\text{-greedy}(\boldsymbol{s}_{\text{ext,sim}}, \pi(\boldsymbol{s}_{\text{ext}}))$
14:            Simulate state transition, $\boldsymbol{s}'_{\text{ext,sim}} \leftarrow M(\boldsymbol{s}_{\text{ext,sim}}, \boldsymbol{a}_{sim})$
15:            Get simulated reward, $r \leftarrow R(\boldsymbol{s}'_{\text{sim}}, \boldsymbol{a}_{\text{sim}})$
16:            Update current state $\boldsymbol{s}_{\text{ext,sim}} \leftarrow \boldsymbol{s}'_{\text{ext,sim}}$
17:            Store tuple in trajectory, $T \leftarrow (\boldsymbol{s}_{\text{ext,sim}}, \boldsymbol{a}_{\boldsymbol{sim}}, \boldsymbol{s}'_{\text{ext,sim}}, r)$
18:        end for
19:        Batch update $Q_{1/2}(\boldsymbol{s}_{\text{ext}}, \boldsymbol{a})$ and $\pi(\boldsymbol{s}_{\text{ext}})$ using $T$
20:    end for
21: end while
```

it sample efficient. TD3 is built upon the deep deterministic policy gradient (DDPG) framework and is designed to address the overestimation bias and function approximation errors commonly encountered in reinforcement learning with continuous action spaces. TD3 introduces several critical improvements: the use of a pair of critic networks to mitigate overestimation by taking the minimum value from the two critics as the target, the policy update delay which decouples the actor and critic updates to prevent destructive interference, and the application of target policy smoothing through adding noise to the target action in the policy update to reduce variance. These improvements make TD3 significantly more stable and reliable, leading to better performance in various continuous control tasks [14].

Within TD3, a pair of critics each individually maintain a function approximation of the state-action value, $Q(\boldsymbol{s}, \boldsymbol{a})$, which estimates the expected future cumulative reward from taking action $\boldsymbol{a}$ while in state $\boldsymbol{s}$. This increases the number of unique neural networks from two to three. Note that this implementation also employs the use of target networks for the actor and two critics. These target networks are clones of the original networks but are updated less often and by copying the parameters of the original networks. Target networks are signified with a $\prime$ in the subscript.

Typically, the target state action value, Q_{target}, is calculated as

$$Q_{\text{target}}(\boldsymbol{s}, \boldsymbol{a}) = R(\boldsymbol{s}, \boldsymbol{a}) + \gamma * Q(\boldsymbol{s}', \boldsymbol{a}') \tag{20}$$

with the next action, $\boldsymbol{a}'$, determined using the policy, $\boldsymbol{a}' = \pi(\boldsymbol{s}')$. The policy is determined by picking the action that maximizes the Q-value

$$\pi(\boldsymbol{s}) = \max_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a}) \tag{21}$$

In TD3, the minimum of the two critics is used

$$Q_{\text{target}}(\boldsymbol{s}, \boldsymbol{a}) = R(\boldsymbol{s}, \boldsymbol{a}) + \gamma \min[Q_{1,}(\boldsymbol{s}', \boldsymbol{a}'), Q_{2,}(\boldsymbol{s}', \boldsymbol{a}')] \tag{22}$$

By taking the minimum, we avoid overestimating the Q-value, leading to better stability.

We also modify the way $\boldsymbol{a}'$ is determined by adding clipped noise

$$\boldsymbol{a}' = clip(\pi_{,}(\boldsymbol{s}') + n, -\boldsymbol{c}_{action}, \boldsymbol{c}_{action}) \tag{23}$$

$$n \sim clip(\mathcal{N}(0, \sigma) * \boldsymbol{c}_{action}, -\delta, \delta) \tag{24}$$

where δ is the noise clip and σ is the policy noise. For this implementation, δ was set to:

$$\delta = 0.5 * \boldsymbol{c}_{action} \tag{25}$$

The addition of noise to the action (when updating the critics) smoothens the Q-value estimate and reduces the chance of the critic networks overfitting to function approximation errors. [14]

The last innovation of the TD3 algorithm is to update the policy network less often than the value network, with one policy update for every four value updates in this implementation. This is to further stabilize the policy. This is also the frequency the target networks are updated. Further details regarding the TD3 algorithm can be found in Ref. [14].

C. Trajectory Sampling

During the inner planning loop of Dyna, the learned state transition model simulates experiences that can be used to update the agent's policy and estimated state-action value functions between real-world actions. In selecting what experiences to simulate, we use a simple trajectory sampling approach where trajectories are simulated starting from the real-world state of the agent. For each trajectory, we start at the current state $\boldsymbol{s}$ and propagate the trajectory forward for l planning steps. At each step, an action, $\boldsymbol{a}_{\text{sim}}$, is chosen using an epsilon-greedy policy based on the current simulation state, $\boldsymbol{s}_{\text{sim}}$. The new simulation state is determined using the transition model of the learned state, $\boldsymbol{s}'_{\text{sim}} \leftarrow M(\boldsymbol{s}_{\text{sim}}, \boldsymbol{a}_{\text{sim}})$, and the corresponding reward is determined from the reward function, $r \leftarrow R(\boldsymbol{s}'_{\text{sim}}, \boldsymbol{a})$. After propagating for l planning steps, the trajectory is used to batch update the state-action value function and policy using TD3. The trajectory sampling is repeated for a total of n trajectories, where the number of planning trajectories, n, and trajectory depth, l, become hyperparameters of the algorithm.

The choice for n determines the number of updates done to the actor and critics. Lower values of n lead to slower learning and longer response times to model changes. Larger values of n allow the network to react quicker but are much more costly to compute, reducing the maximum control frequency. On the other hand, the choice for l affects the amount of exploration the agent is able to gather. Since the actions in the planning steps are determined using an ϵ-greedy algorithm, larger l values create more divergent trajectories. Additionally, larger values of l allow for the agent to see further, allowing for it to avoid state violations. Increasing l also increases computational time, but this has less of an effect than n, allowing for the use of larger l values when running on flight-like hardware.

D. Transition Model

The transition model, $M(\boldsymbol{s}, \boldsymbol{a})$, is the agent's internal model of the real environment dynamics. The objective of the transition model is to accurately determine the next state, $\boldsymbol{s}'$, given a state-action pair, $\boldsymbol{s}$ and $\boldsymbol{a}$. In the Dyna architecture, the transition model is only updated from real-world actions and is not updated during the planning loop. In our implementation, the transition model consists of a single-layer perceptron (SLP). The purpose of the SLP is to determine the next state given a state-action pair. The SLP is initially trained offline using the nominal system dynamics outlined in Section II.A. During the mission, the measured $\boldsymbol{s}'$ are used to update the SLP using backpropagation. The loss is calculated as follows:

$$loss_t = MSE(\boldsymbol{s}' - M(\boldsymbol{s}, \boldsymbol{a})) \tag{26}$$

where MSE is the mean-squared error. This allows the agent to adapt to changing dynamics in real time.

E. Learning Rate Suppression

Figure 2 shows the transition loss, $loss_t$, over the course of an episode with a thruster failure at t = 0s. The spike in the loss and its subsequent decay is due to the agent actively relearning the model. During this process, the outputs of $M(\boldsymbol{s}, \boldsymbol{a})$ (used inside of the planning steps) may be inaccurate, and using these values could add instability to the actor and critics. To counteract this, while the agent is relearning the model, signified by a transition loss greater than some threshold Γ, we lower the learning rate, α, of both the actor and critic. In our implementation, this is done as follows:

$$\alpha = \begin{cases} 0.001 & loss_t < \Gamma \\ 0 & loss_t \geq \Gamma \end{cases} \tag{27}$$

where $\Gamma = 0.0001$ in our implementation.

As a general note, α is *not* modified for the actor and critic update in Algorithm Step 6, which uses the observed next state, $\boldsymbol{s}'$, rather than the simulated output ($\boldsymbol{s}'_{sim}$) from $M(\boldsymbol{s}, \boldsymbol{a})$.

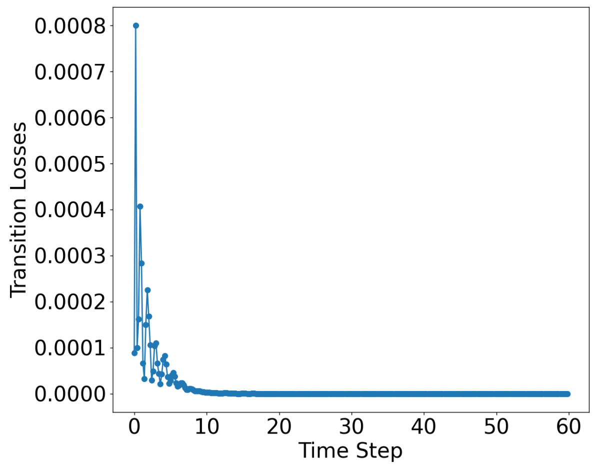

Fig. 2 Transition loss with a single thruster failure at t = 0s

Table 1 TD3-Dyna Neural Network Structures

	Actor	**Critics**	**Transition Model**
Input	s_{ext}	$s_{ext}, \boldsymbol{a}$	$s_{ext}, \boldsymbol{a}$
Output	$\boldsymbol{a}$	Q	$\boldsymbol{s}'_{ext}$
Shape	(64,64)	(64,64,64)	(32)
Activation Functions	ReLU with Tanh on output layer	ReLU	SiLU

IV. Training and Development

Our Dyna implementation has four main neural networks that need to be trained: the actor, the two critics, and the transition model (we exclude the target networks as they are direct clones of these networks). Rather than train all four together as in a typical reinforcement learning training scheme, we train each separately to reduce training time and build in specific properties that decrease the unpredictability of the agent. The structure of each network is given in Table 1. As a general note, the two critics are initialized to be identical (with identical shape and initial values).

For the actor and critic, the Rectified Linear Unit (ReLU) activation function was used due to its simplicity. ReLU is defined as

$$\text{ReLU}(x) = \max(0, x) \tag{28}$$

However, the transition model uses a variant of ReLU called the Sigmoid Linear Unit (SiLU), also known as the Sigmoid-Weighted Linear Unit or Swish. It is defined as

$$\text{SiLU}(x) = \frac{x}{1 + e^{-x}} \tag{29}$$

SiLU smoothly interpolates between a linear activation and a non-linear activation, providing a balance that can improve performance in some neural networks [15]. The choice to use SiLU over ReLU for the transition model was to improve the robustness of the transition model to a variety of inputs. Additionally, the simplicity of the network shape (only one hidden layer) and its relative infrequency of updates (only once per time step) ensures the increased computational complexity from using SiLU is not noticeable.

A. Actor

The actor, or the policy, is responsible for providing the optimal control action for a given state. Under nominal, failure-free, dynamics, classical control algorithms work quite well. Therefore, we can train the actor using imitation learning with the traditional PD-LQR controller described in Section II.C as the "expert" [16]. With the PD-LQR feedback controller, we generate 10^5 random state vectors such that

$$|\boldsymbol{s}| \leq \begin{bmatrix} 7\text{ m} & 7\text{ m} & \pi & 5\text{ m/s} & 5\text{ m/s} & 10\text{ rad/s} \end{bmatrix}^T \tag{30}$$

The range of random states extends beyond the states that are members of $\{\chi_{\text{safe}}\}$ or are expected to be observed, this is done to eliminate performance degradation arising from experiencing states far outside of the training set, as neural networks tend to struggle with generalization [17].

The optimal control actions are determined using

$$\boldsymbol{a}_{\text{optimal}} = -[K]\boldsymbol{s} \tag{31}$$

We then fit our neural network to this dataset using mean squared error as our loss function over 75 epochs of batch size 100. The neural network structure for the actor was obtained by minimizing the loss function using trial and error. It should be noted that the optimal actor structure contains no hidden layers or activation functions. However, this was not chosen as it constrains the agent to linear policies, which would perform poorly should the optimal policy become nonlinear due to unexpected model changes.

B. Transition Model

Similar to the actor, we generate a dataset of random states and actions and calculate the next state, $\boldsymbol{s}'$, using Equation 4. Although the dynamics are known and linear (and would not necessitate a neural network of this complexity), thruster failures or model changes would lead to definitively nonlinear dynamics that cannot be captured using simpler neural network structures.

C. Critic

In Q-learning, the Q-value is updated according to

$$Q^{new}(\boldsymbol{s}, \boldsymbol{a}) \leftarrow Q(\boldsymbol{s}, \boldsymbol{a}) + \alpha \left[r + \gamma \max_{a}^{\prime} Q(\boldsymbol{s}', \boldsymbol{a}') - Q(\boldsymbol{s}, \boldsymbol{a}) \right] \tag{32}$$

Since the formula for $Q(\boldsymbol{s}, \boldsymbol{a})$ depends on the Q for a different state-action pair, it is difficult to train the critic in the same manner as the transition and actor models. Therefore, we converge the critic model using a traditional RL training loop, as outlined in Algorithm 2.

Algorithm 2 Critic Training Algorithm

```
1: Initialize Q-function $Q(\boldsymbol{s}, \boldsymbol{a})$, optimal policy $\pi^*(S)$ and model $M(s, a)$
2: for $n$ episodes do
3:     $\boldsymbol{s} \leftarrow$ Get random initial state
4:     while not done do
5:         $\boldsymbol{s} \leftarrow$ current agent state
6:         $\boldsymbol{a} \leftarrow \epsilon$-greedy$(S, \pi^*)$
7:         Take action $\boldsymbol{a}$, observe new state $\boldsymbol{s}'$, reward $r$, and terminal condition done
8:         Update $Q(\boldsymbol{s}, \boldsymbol{a})$
9:         $\boldsymbol{s} \leftarrow \boldsymbol{s}'$
10:    end while
11: end for
```

To encourage exploration and improve the accuracy of the Q value in suboptimal states, we use a high ϵ (0.25) in our greedy epsilon strategy. An important consideration when training using this method is to balance the use of the optimal policy with random actions. Due to the high dimensionality of the input, it is impossible to explore every possible state-action pair. Therefore, we must approximate the Q-value using a neural network. Only using the optimal policy during critic training leads to an overrepresentation of "good" outcomes and high Q-values, leading to a critic that is overly optimistic and unstable, while only relying on random actions leads to an over-representation of lower Q-values and an overly pessimistic critic function.

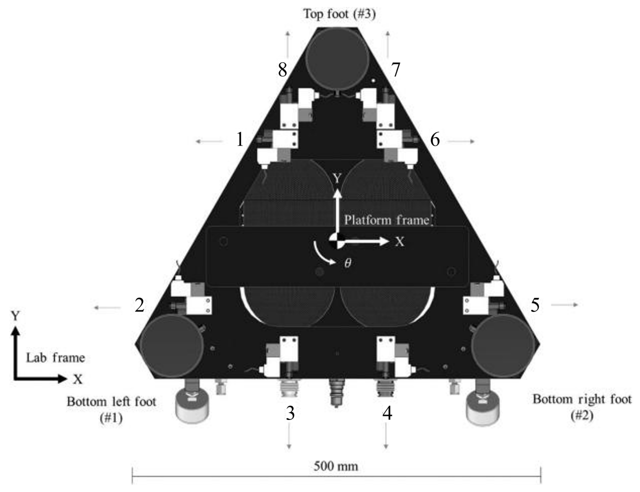

Fig. 3 Planar air-bearing thruster diagram, adapted from Ref. [11]

V. Evaluations

The Dyna agent was evaluated in both simulation and hardware environments. Hardware testing was conducted at NASA/JPL's GSAT (GNC (Guidance, Navigation, and Control) Spacecraft Autonomy Testbed) using a planar air-bearing sled [11]. The Dyna agent was compared against a PD controller using LQR gains determined in Section II.C. The performance cost for both agents was evaluated according to

$$J = \sum_{i=0}^{k} (\boldsymbol{s}_i^T [Q] \boldsymbol{s}_i + \boldsymbol{a}_i^T [R] \boldsymbol{a}_i) \Delta t \tag{33}$$

where k is the total number of time steps; $k = 300$ for all test cases for a total simulation time of 60 s and a time step of 0.2 s. This cost function is the same as the reward function given in Equation 13 without terminal state penalties.

A. Hardware Setup

The hardware testing was performed on the planar air bearing platform located in the GSAT (GNC(Guidance, Navigation, and Control) Spacecraft Autonomy Testbed) at NASA's Jet Propulsion Laboratory, shown in Figure 4. The planar air-bearing platform consists of a 2D flat floor and a platform that maintains a reduced-friction environment by expelling air through planar air bearings. The platform is able to move along two translational degrees of freedom and one rotational degree of freedom (whose axis of rotation is normal to the plane). The air-bearing platform consists of eight thrusters, which can be controlled independently to achieve the desired attitude [11]. Figure 3 shows the thruster layout of the planar air-bearing platform.

The controller outputs high-level actions in the form of the desired net forces and torques given in the reference frame. These actions need to be converted to individual thruster commands to simulate thruster degradation and fire the physical thrusters. On the planar air-bearing platform, thrust allocation is formed using a mixer matrix, $[M]$. The individual thruster forces are then calculated by solving the following constrained minimization problem using least squares

$$\min |\boldsymbol{a} - [M]\boldsymbol{u}| \tag{34}$$

$$\text{subject to } |\boldsymbol{u}| \geq 0 \tag{35}$$

where $\boldsymbol{u}$ is a vector of requested individual thruster forces. Further details on the formation of $[M]$ and the thrust allocation can be found in Ref. [11]. Since this method of thrust allocation is blind to hardware thrust constraints, it is possible that the requested force and the actual force differ by a significant margin. This also makes the true dynamics of the system nonlinear.

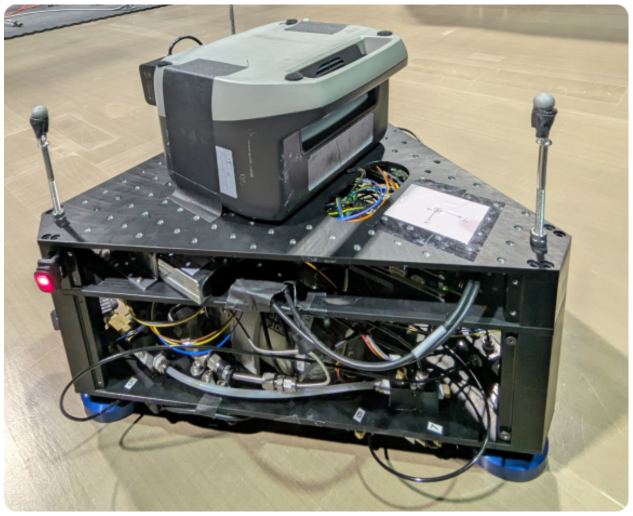

Fig. 4 The air bearing sled on JPL's GSAT flat floor facility used for hardware verification, with +y pointing toward the right.

B. Handling Failure

To model failure, we modify the thruster level control input as

$$\boldsymbol{u}_i^* = \eta \boldsymbol{u}_i \tag{36}$$

where $\boldsymbol{u}_i^*$ is the modified thrust output for the thruster, $\boldsymbol{u}_i$ is the required thrust output for the thruster i and η is the degradation parameter. For each simulation, η is randomly determined from a uniform distribution ranging from 0 to 1. Physically, when $\eta = 1$, the thruster is functioning as expected (no failure) and a $\eta = 0$ is equivalent to a stuck-off failure (no thrust output). To keep testing simple and easily reproducible on hardware, the failures always occurred at the beginning of the simulation (t = 0 s). It should also be noted that neither the controller nor the thrust allocator are aware of the degraded thruster and are still able to request a force from the faulty thruster. This means the agent must identify the thruster failure through the state measurements and react accordingly.

C. Simulation Results

The controllers were evaluated in a Monte Carlo simulation of 71,000 trials. The initial states were randomly generated but kept consistent between the different controllers. Additionally, the models were reset after each episode to prevent the agent from learning from previous episodes. Figure 5 shows the average cost for both the Dyna agent and PD-LQR controllers as a function of the degradation factor η. To determine the average cost, the data was divided into 20 equally spaced "buckets" based on the degradation factor. The average cost for all trials in each bucket was then calculated and plotted. Furthermore, all trials that contained a failure (where a controller exceeded state bounds) in either agent were excluded from the dataset used to generate Figure 5. This exclusion was to prevent low costs from early episode terminations from influencing the dataset. Figure 5 shows that Dyna and PD-LQR perform similarly at high η values, but Dyna tends to perform better at lower η values. The large error bars are due to the high variance in episode costs caused by initial state and failure modes.

Figures 6 and 7 show the median cost difference between Dyna and the PD-LQR controller, as well as the probability that Dyna has a lower cost than PD-LQR. We observe that, on average, Dyna outperforms PD-LQR, regardless of the degradation factor. The error bars for the cost difference increase slightly as η approaches 0 because the outcome of an episode heavily depends on the initial state and the specific thruster that failed. For example, a degraded thruster may not be used much for a given initial state, increasing the likelihood that the PD-LQR controller outperforms Dyna. Conversely, if the failed thruster is heavily used by PD-LQR for a certain initial state, then Dyna is more likely to outperform the other controller.

Figure 8 shows the probability that either controller fails the regulation task (exceeds the state constraints). We see that the probability of failure increases as η decreases and that Dyna generally has a lower failure rate than PD-LQR,

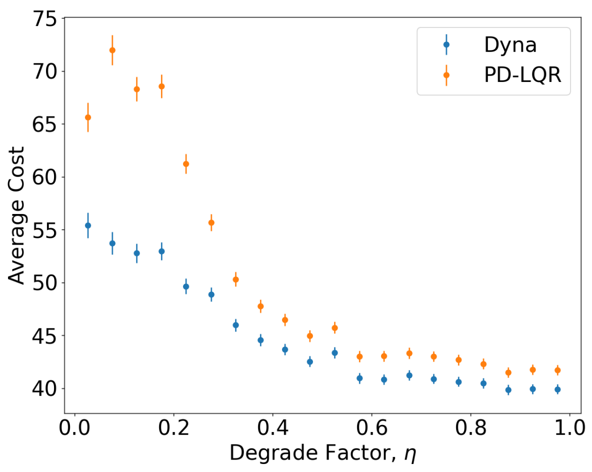

Fig. 5 Average episode cost for Dyna and PD-LQR. Episodes where a controller exceeded state bounds are removed.

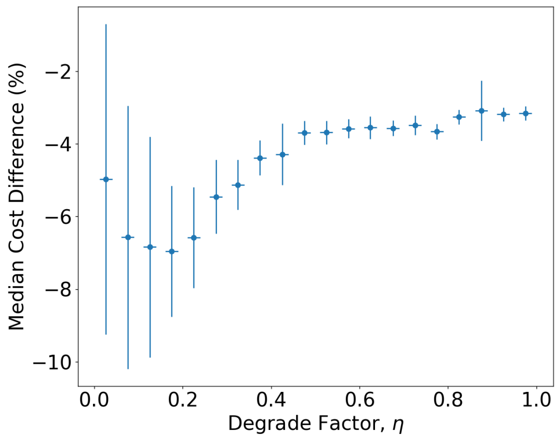

Fig. 6 Median cost difference between Dyna and PD-LQR. Cost difference is calculated as $J_{\text{Dyna}} - J_{\text{LQR}}$. Episodes where a controller exceeded state bounds are removed.

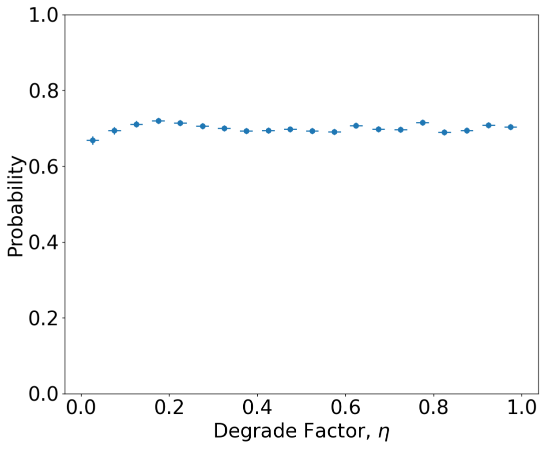

Fig. 7 Probability that $J_{\text{Dyna}} < J_{\text{LQR}}$. Episodes where a controller exceeded state bounds are removed.

demonstrating that Dyna is able to adapt to the degraded model and still complete the task.

D. Hardware Results

The controllers were run at a 5Hz control rate and implemented inside of the F Prime (F') framework running on a Raspberry Pi 5. F' is a flight software and embedded systems framework that has been used on several space missions including a number of SmallSats and CubeSats such as those Steward might target [18]. The Raspberry Pi 5 is a common single-board computer used as the main flight computer for many CubeSats and SmallSats and offers enough computational power to run high-level controllers at a real-time frequency. Both controllers started from the same initial state and ran for 60 seconds. Each initial state was repeated three times and the average was taken. Additionally, all initial states during hardware testing began with no initial angular or translational velocity to ensure the initial states for across controllers and trials were as similar as possible.

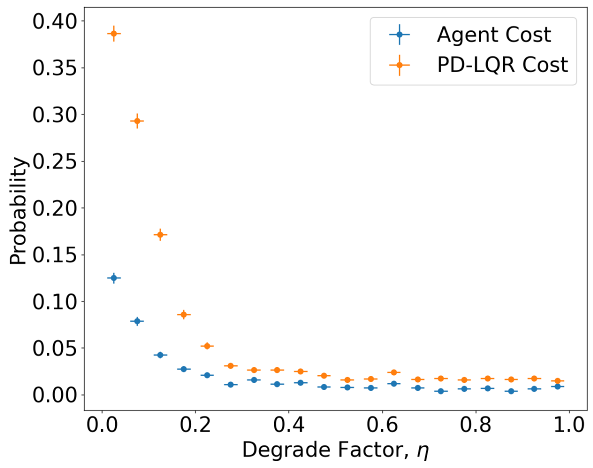

Fig. 8 Probability that a controller exceeds state bounds.

Table 2 PD-LQR and Dyna cost comparison (failure-free)

Controller	Trial 1	Trial 2	Trial 3	Experimental Average	Simulation
Dyna	25.78	29.20	30.06	28.35 ± 1.07	12.05
PD-LQR	38.96	32.13	21.90	30.99 ± 4.05	8.28

1. The Failure-Free Case

We begin our analysis with the failure-free case. Table 2 shows the calculated cost for both controllers across all trials along with the simulation prediction and the calculated average. We see that Dyna generally outperforms PD-LQR in hardware testing (there is some overlap in error) but the opposite is true in simulation. We also see that the hardware cost is higher than the simulation cost, this is mainly due to the difference between hardware and simulation testing. For example, irregularities in the flat floor or inaccuracies in state measurements can manipulate the dynamics of the sled.

2. Single Thruster Failure

On hardware, thruster failures were achieved by disconnecting the thruster valve from power, ensuring the valve always remained closed and unresponsive to controller commands. This allows for the controller and thrust allocator to still request forces from the failed thruster (but it outputs no force). Table 3 again shows the calculated cost from this position for both experiment and simulation. We see that PD-LQR outperforms Dyna with a single thruster failure in both hardware and simulation. This is also an expected result, as the simulation results in Figure 7 show a significant chance that PD-LQR outperforms Dyna when dealing with a stuck-off failure. This is mostly because the effect of a single stuck-off thruster is highly dependent on the initial state.

However, there is more to the story. PD controllers typically suffer from steady-state error. This arises because the errors become too small for the controller to provide any meaningful corrective action. This error increases when the true dynamics begin to differ from the expected dynamics. Figure 9 shows the trajectories for the single-failure case for both PD-LQR and Dyna controllers.

Table 3 PD-LQR and Dyna Cost Comparison (thruster 7 stuck off)

Controller	Trial 1	Trial 2	Trial 3	Experimental Average	Simulation
Dyna	31.57	28.86	43.82	34.75 ± 3.76	12.39
PD-LQR	18.77	18.29	19.46	18.84 ± 0.27	8.55

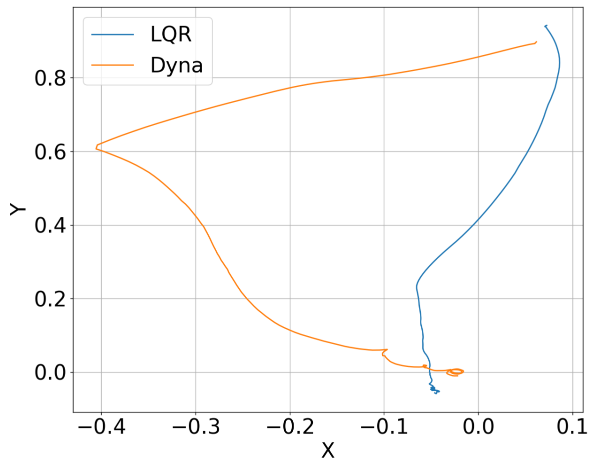

Fig. 9 Trajectory comparison for single failure case between Dyna and PD-LQR controllers.

Table 4 PD-LQR and Dyna cost comparison (thrusters 2 and 4 stuck off).

Controller	Trial 1	Trial 2	Trial 3	Experimental Average	Simulation
Dyna	32.90	34.62	29.93	32.48 ± 1.12	84.48
PD-LQR	24.04	26.50	21.38	23.98 ± 1.21	15.9

We clearly see the presence of a steady state error for the LQR controller due to model degradation, as expected. However, the Dyna controller is able to correct for this steady-state error. This is promising, as the agent was trained *using* the exact same PD-LQR controller used meaning the agent has learned to overcome this deficiency in the PD-LQR controller and still stabilize the spacecraft.

3. Double Thruster Failure

We now compare the performance of these controllers with a significant model degradation, as represented by two thruster failures (Thrusters 2 and 4). Table 4 shows the average cost for each controller. We see that PD-LQR outperforms Dyna again according to the LQR cost. While the simulation result for Dyna overshoots the expected cost, the experiment outcome matches the simulation predictions.

Like in Section V.D.2, we see that Dyna is able to eliminate the steady-state error, as shown in Figure 10.

4. Dyna Runtime Analysis

Table 5 shows the average runtimes for the main functions in the algorithm. The `getAction()` function involves retrieving an action from the agent's policy (Step 4 in Algorithm 1). The `learn()` function is responsible for performing updates to the neural network and includes all the planning steps in Dyna (steps 5–19 in Algorithm 1). From Table 5, it is evident that Dyna is capable of operating in real-time on flight-like processors. The `getAction()` function is fast, with mean runtimes close to 1 ms (1000 Hz). The high standard deviation is due to background processes running on the Raspberry Pi. The `learn()` function takes significantly longer and is the limiting factor for the maximum control frequency at which Dyna can operate. The runtime for `learn()` is linearly dependent on the number of trajectories simulated in the planning steps, as this number also determines the number of backward passes through each neural network.

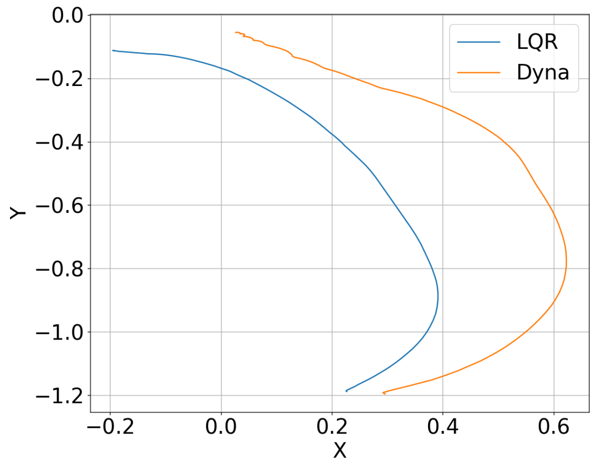

Fig. 10 Trajectory comparison for the double failure case between Dyna and PD-LQR controllers.

Table 5 Dyna key function runtimes.

Function	**Mean Runtime (ms)**	**Standard Deviation (ms)**
getAction()	0.959 ± 0.0032	6.462
learn(), $n = 5$, $l = 20$	100.587 ± 0.0044	8.761

VI. Conclusion

Deployable free-flying small spacecraft represent a promising solution for exterior inspection of future space stations, addressing the constraints of current techniques employed on the International Space Station. By autonomously conducting regular or on-demand inspections, these spacecraft can cover the entire exterior surface and minimize the need for human involvement or costly extra-vehicular activities. Nonetheless, ensuring the safety and reliability of free-flying spacecraft during close proximity operations is crucial to prevent potential collisions with the space station. Existing strategies that rely on onboard thrusters for collision-avoidance maneuvers face challenges due to variable thruster performance, which can hinder the ability to execute precise maneuvers. This variability, coupled with the computational demands of traditional nonlinear trajectory optimization, necessitates the development of more adaptive and computationally efficient solutions.

Our proposed reactive and integrated architecture, SmallSat Steward, harnesses the Dyna reinforcement learning framework to address these challenges. By combining model-based planning and direct reinforcement learning, Dyna offers a flexible, computationally efficient solution capable of adapting to changes in thruster performance and other system uncertainties. Preliminary results in both simulation and hardware environments demonstrate the potential of this architecture to maintain stable operations even under significant model degradation. Future work will focus on refining the SmallSat Steward architecture, extending evaluation to more complex failure modes, incorporating more realistic orbital mechanics (e.g. Clohessy-Wiltshire dynamics), and further testing on flight-like hardware.

VII. Acknowledgements

Funding for this work was provided by the NASA Space Technology Mission Directorate through the University SmallSat Technology Partnerships program under grant 80NSSC23M0237. Parts of this work were carried out at the Jet Propulsion Laboratory, California Institute of Technology under contract 80NM0018D0004 with the National Aeronautics and Space Administration.

References

[1] Walton, P. et al., "Passive CubeSats for Remote Inspection of Space Vehicles," *Journal of Applied Remote Sensing*, Vol. 13, No. 3, 2019, p. 032505. https://doi.org/10.1117/1.JRS.13.032505.

[2] Fredrickson, S. E., Duran, S., Howard, N., and Wagenknecht, J. D., "Application of the Mini AERCam Free Flyer for Orbital Inspection," *Proceedings of Defense and Security*, Society of Photo-Optical Instrumentation Engineers (SPIE), Orlando, Florida, USA, 2004. https://doi.org/10.1117/12.542810.

[3] Pedrotty, S., Sullivan, J., Gambone, E., and Kirven, T., "Seeker Free-Flying Inspector GNC Flight Performance," *Proceedings of the AAS Guidance and Control Conference*, American Astronautical Society (AAS), Breckenridge, Colorado, USA, 2020.

[4] Jacklin, S. A., "Small-Satellite Mission Failure Rates," NASA Technical Report, NASA Ames Research Center, Moffett Field, CA, 2019.

[5] Pedrotty, S., Sullivan, J., Gambone, E., and Kirven, T., "Seeker Free-Flying Inspector GNC System Overview," *Proceedings of the AAS Guidance and Control Conference*, American Astronautical Society (AAS), Breckenridge, Colorado, USA, 2019.

[6] Corpino, S. and Stesina, F., "Inspection of the Cis-Lunar Station using Multi-Purpose Autonomous CubeSats," *Acta Astronautica*, Vol. 175, 2020, pp. 591–605. https://doi.org/10.1016/j.actaastro.2020.05.053.

[7] Gambone, E. A., "Seeker CubeSat Control System," *Proceedings of the AIAA SciTech Forum*, American Institute of Aeronautics and Astronautics (AIAA), Orlando, Florida, USA, 2020. https://doi.org/10.2514/6.2020-1102.

[8] Smith, C. et al., "The Journey of the Lunar Flashlight Propulsion System from Launch through End of Mission," *Small Satellite Conference*, Utah State University, Logan, Utah, USA, 2023.

[9] Albee, K., Ekal, M., Coltin, B., Ventura, R., Linares, R., and Miller, D. W., "The RATTLE Motion Planning Algorithm for Robust Online Parametric Model Improvement With On-Orbit Validation," *IEEE Robotics and Automation Letters*, Vol. 7, No. 4, 2022, pp. 10946–10953. https://doi.org/10.1109/LRA.2022.3196957.

[10] Sutton, R. S., "Dyna, an Integrated Architecture for Learning, Planning, and Reacting," *ACM SIGART Bulletin*, Vol. 2, No. 4, 1991, pp. 160–163. https://doi.org/10.1145/122344.122377.

[11] Wapman, J. D., Sternberg, D. C., Lo, K., Wang, M., Jones-Wilson, L., and Mohan, S., "Jet Propulsion Laboratory Small Satellite Dynamics Testbed Planar Air-Bearing Propulsion System Characterization," *Journal of Spacecraft and Rockets*, Vol. 58, No. 4, 2021, pp. 954–971. https://doi.org/10.2514/1.A34857.

[12] Albee, K., Sternberg, D., Hansson, A., Schwartz, D., Majumdar, R., and Jia-Richards, O., "Architecting Autonomy for Safe Microgravity Free-Flyer Inspection," *IEEE Aerospace Conference*, IEEE, Big Sky, MT, USA, 2025.

[13] Geist, A. R., Frey, J., Zobro, M., Levina, A., and Martius, G., "Learning with 3D rotations, a hitchhiker's guide to SO(3)," , 2024. URL https://arxiv.org/abs/2404.11735.

[14] Fujimoto, S., van Hoof, H., and Meger, D., "Addressing Function Approximation Error in Actor-Critic Methods," *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 80, edited by J. Dy and A. Krause, PMLR, 2018, pp. 1587–1596. URL https://proceedings.mlr.press/v80/fujimoto18a.html.

[15] Ramachandran, P., Zoph, B., and Le, Q. V., "Searching for Activation Functions," , 2017. URL https://arxiv.org/abs/1710.05941.

[16] Zare, M., Kebria, P. M., Khosravi, A., and Nahavandi, S., "A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges," , 2023. URL https://arxiv.org/abs/2309.02473.

[17] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R., "Intriguing Properties of Neural Networks," *arXiv preprint arXiv:1312.6199*, 2013.

[18] Bocchino, R. L. J., Canham, T. K., Watney, G. J., Reder, L. J., and Levison, J. W., "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems," , 2018. https://doi.org/2014/48425, URL https://hdl.handle.net/2014/48425.