

Safe Reinforcement Learning Benchmark Environments for Aerospace Control Systems

Umberto J. Ravaioli
Toyon Research Corp.
6800 Cortona Dr.
Goleta, CA, 93117
uravaioli@toyon.com

Vardaan Gangal
Jacobs Engineering Group
1415 Research Park Dr.
Beavercreek, OH 45432
vardaan.gangal@jacobs.com

James Cunningham
Jacobs Engineering Group
1415 Research Park Dr.
Beavercreek, OH 45432
james.cunningham@jacobs.com

Kyle Dunlap
University of Cincinnati
2901 Woodside Drive
Cincinnati, OH, 45219
dunlapkp@mail.uc.edu

John McCarroll
Matrix Research Inc.
607 Washington St.
Newton, VT 02458
john.mccarroll.ext@afresearchlab.com

Kerianne L. Hobbs
Air Force Research Laboratory
2241 Avionics Circle
Wright-Patterson Air Force Base, OH, 45433
kerianne.hobbs@us.af.mil

Abstract—Recent advancements in reinforcement learning techniques demonstrate an ability to make decisions in high dimensional state spaces and complex real-time strategy games. In contrast to supervised learning which features large data sets, there are relatively few existing environments for training reinforcement learning agents. In addition, small differences in rewards or action spaces can drastically change the difficulty and results of the training environments. Benchmarks seek to tackle both of these challenges by creating common environments, in the form of “Gyms” to train and compare reinforcement learning techniques, approaches, and algorithms. Many gyms, such as the classical control and Atari games environments, have become standard in new research on reinforcement learning. Researchers can easily compare and benchmark competing solutions across publications on these universal baselines enabling rapid innovation and collaboration. However, there are currently no standard set of environments for aerospace problems, and many of the gyms in the literature do not include safety constraints or run time assurance systems that intervene when the reinforcement learning agent violates safety constraints. This manuscript describes the development of the Aerospace SafeRL Framework and accompanying Aerospace SafeRL Benchmarks that include interactive environments, safety constraints, software interfaces for run time assurance safety monitors with base implementations, and an initial set of baseline solutions. This initial set of scenarios introduces simple RL environments that expose the kinds of motion patterns, dynamics, and safety constraints encountered in air and space problems in 2D and 3D. This manuscript also describes standardized evaluation metrics for these environments to provide a consistent performance measurement with aerospace relevance. These benchmarks provide a structured foundation for future reinforcement learning algorithms, run time assurance designs, and neural network verification techniques for the aerospace domain.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. REINFORCEMENT LEARNING FOR CONTROL ...	2
3. ENVIRONMENTS	4
4. AEROSPACE SAFERL ARCHITECTURE	6
5. METRICS	9
6. BASELINE PERFORMANCE	10
7. CONCLUSIONS AND RECOMMENDATIONS	12
APPENDIX	13
ACKNOWLEDGMENTS	17

REFERENCES	19
BIOGRAPHY	20

1. INTRODUCTION

Reinforcement Learning (RL) has recently demonstrated prowess that surpasses humans in both high dimensional decision spaces like Go [1] and in complex real-time strategy games like StarCraft [2]. In aerospace, RL could provide new solution spaces for complex control challenges in areas ripe for autonomy such as urban air taxis, package delivery drones, satellite constellation management, in-space assembly, and on-orbit satellite servicing. Generally, advancements in machine learning technology, including RL, have depended on common frameworks and benchmarks to facilitate collaboration and competition within the research community. The majority of RL research focuses on a standard set of benchmark environments, such as toy control problems or Atari Games [3], that excludes safety critical aerospace problems and applications. While there have been a few papers applying RL to hypothetical aircraft and spacecraft, the environments are ad hoc and do not feature significant investment from the AI, academic, or industrial community. In contrast to video games, and therefore much of the existing corpus of RL research, aircraft and spacecraft are safety and mission critical systems, placing aerospace solutions in the category of SafeRL [4]. In aerospace applications, a poor decision from an RL agent could result in a loss of life in the air domain or loss of a highly valuable space-based service in the space domain. Although some progress has been made towards benchmarks for safety [5], the environments do not feature run time assurance (RTA), which limits the current research.

This paper documents a new effort to close the application gap in RL research by providing air and space benchmark environments, including safety constraints and RTA architectures, within in the Aerospace SafeRL Framework. Implementing the OpenAI Gym API [3], these environments are compatible with most RL training frameworks and provide a common testing ground for aerospace-centric techniques and solutions. Each environment provides an interactive simulation for RL agents to experiment on and learn solutions. In each domain, multiple dynamics models of varying detail are available that provide differing levels of abstraction or difficulty to the agent. In the air domain, the *rejoin* task features a wingman aircraft agent learning to follow a lead aircraft in formation flight. In the space

domain, a *docking* task features a deputy spacecraft agent learning to dock with a chief spacecraft while they orbit a third central body. Additionally, each environment includes a variety of safety constraints, such as velocity limits and safe separation requirements. The ability of RL solutions to comply with these constraints while completing the primary task is measured and quantified. Flexible software modules allow custom configurable reward shaping and observation formatting allowing research teams to rapidly explore possible solutions and problem limitations. By remixing and extending the modular models and components, new environments that address the future challenges of aerospace RL can be built. Thus, the Aerospace SafeRL Framework provides the community with the common ground it needs to advance RL within the aerospace domain. This paper describes the environments, setup of the framework, and example training results.

The rest of this paper is organized as follows. First, an introduction to RL concepts applied to controls, SafeRL approaches including RTA, and considerations for metrics to evaluate SafeRL approaches are discussed. Second, each of the five initial environments in the SafeRL framework are described. Third, an overview of the Aerospace SafeRL framework architecture is provided. Fourth, metrics used to evaluate the framework are presented. Fifth, baseline performance of the benchmarks is presented. Finally, conclusions and recommendations are made for follow on research.

2. REINFORCEMENT LEARNING FOR CONTROL

This research explores the use of SafeRL in aircraft and spacecraft control. This section briefly compares RL to traditional control theory concepts, describes RL agent learning, introduces the default RL library used in the framework, discusses SafeRL approaches, describes RTA approaches, and finishes with considerations for the evaluation metrics selected later in the paper.

Comparing Control Theory and Reinforcement Learning Concepts

In this paper, RL is used in an aerospace control context. Here, the controls theory and RL terms are compared. Where applicable, equivalent control theory and RL terms are listed together separated by a “/” with the controls term coming first. Control theory and RL share a common concept of a system *state*, which are variables that describe the plant / environment such as position and velocity. Both control theory and RL have a notion of *partial observability*, where a measurement / *observation* may not fully describe the system state but could be used to estimate it. A discrete or continuous control input / *action* is determined by a control law / *policy*, which is a function of the current state or partial observation that is optimized to minimize a cost function in controls or maximize a *reward function* in RL. In response to this action, the environment state evolves in accordance to a deterministic or probabilistic state transition function. In Aerospace SafeRL, state transitions are modeled with deterministic physics *dynamics* models, as is typically done in controls.

Reinforcement Learning Agents

RL agents are composed of any or all of the following: a *policy*, *value function*, and *state transition model*. The policy, denoted by π , yields a probability distribution of

Table 1. Comparison of Control Theory and Reinforcement Learning Concepts

Control Theory		Reinforcement Learning	
Plant		Environment	
$x \in \mathcal{X}$	state	$s \in \mathcal{S}$	state
$y \in \mathcal{Y}$	measurement	$o \in \mathcal{O}$	observation
Controller Control Law		Agent Policy (π)	
$u \in \mathcal{U}$	control input	$a \in \mathcal{A}$	action
J	cost function	\mathcal{R}	reward function
$x_{k+1} = f(x_k, u_k)$		$\mathbb{P}(s_{t+1} s_t, a_t)$	

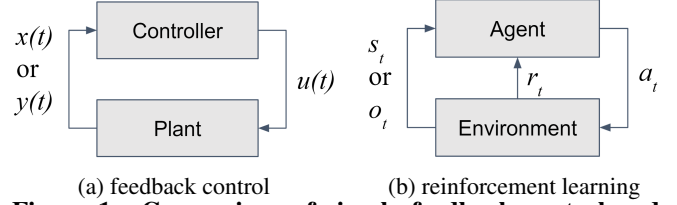


Figure 1. Comparison of simple feedback control and reinforcement learning models

potential agent actions as a function of the current state, $\pi(a|s) = \mathbb{P}(a_t = a|s_t = s)$. It may be stochastic or deterministic. The value function estimates the expected future discounted reward given a state under a given policy, $v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots | s_t = s]$. Similarly, the *state-action value function*, $Q_\pi(s, a)$, estimates the expected future discounted rewards given an action and state under a given policy. Value-based reinforcement learning methods derive their policy by greedily maximizing their value function while policy based methods directly parameterize a policy function. Model based RL methods utilize a state transition model, $\mathbb{P}(s_{t+1}|s_t, a_t)$, to estimate the next state given the current state and policy selected action. Other model-free methods do not explicitly model state transitions and instead rely on only their policy and/or value functions.

Explicitly tabulating these function outputs with large state and/or action spaces is intractable. Deep reinforcement learning addresses this by smoothly approximating the agent component functions with deep neural networks. Each network is composed of a series of layers that sequentially apply a linear transformation, parameterized by weight and bias matrices, and non-linear activation function to the input vector to produce a rich, non-linear output. As the agent interacts with the environment and receives reward feedback, the weights and biases of the network layers are updated via gradient descent optimization to better approximate their target quantities or produce an action that better maximizes expected future discounted reward.

RL agents learn by interacting with and receiving reward feedback from the environment, a representation of the problem task the agent learns to solve. These interactions are typically modeled with computer simulations, but may produced with physical simulations or real task experience. A sequence of environment-agent step *interactions*, formally the tuple of observation, action, and reward at a time t , from the initial state to terminal end state of a task is called an *episode*. Subsequences of episodes are called *trajectories* and trajectories containing terminal states are called *rollouts*, although this nomenclature is flexible and may vary. In each

training *iteration*, the agent interacts with the environment with a frozen policy, fills a training *buffer* with those interaction experiences, and finally samples from said training buffer to perform gradient descent updates. The gradient descent update occurs in mini-batches over K full walks, or *epochs*, of the training buffer.

Reinforcement Learning Library

Aerospace SafeRL and the results presented in this paper are built around the Ray RLlib framework [6] which is designed for scalability and multi-agent RL; however, the environments conform to the community standard Open AI Gym API [3] and can be used with any compliant RL framework. The baseline performance agents presented in this paper are composed of a standard shallow multilayer perceptron policy network with tanh activations trained with Proximal Policy Optimization (PPO) [7]. PPO is an on-policy, actor-critic policy gradient method whose robustness and efficacy has made it the defacto baseline approach for many RL problems including continuous control. As a policy gradient method, it is efficient in tasks with large or continuous action spaces. By clipping the surrogate objective function, PPO limits short term changes to the policy network and greatly reduces catastrophic forgetting, sudden performance collapses caused by excessive training updates common in RL algorithms [8]. While PPO is a suitable choice for these problems, it is possible and encouraged to apply alternative RL algorithms to these benchmark tasks.

Safe Reinforcement Learning

When an RL agent is in control of a physical system, such as a robot, aircraft, or spacecraft, ensuring safety of that agent and the humans who interact with it becomes critically important. SafeRL approaches learn a policy that maximizes reward while adhering to safety constraints [4]. Approaches to SafeRL fall under the category of *reward shaping* which incorporates safety into the reward function, or *shielding* (i.e. RTA) [9], which monitors the RL agent outputs and intervenes by modifying the action to ensure safety. These benchmarks incorporate both approaches. Penalties are given for violating safety during the training process, and the environments are designed to incorporate different RTA designs and penalties for their use during training.

Run Time Assurance Approaches

RTA is an online safety assurance technique that filters potentially unsafe inputs from a primary controller in a way that preserves the safety of the system when necessary. The control system is divided into a performance-driven primary controller and a safety-driven RTA filter as shown in Figure 2, where the components outlined in red have low safety confidence and the components outlined in blue have high safety confidence.

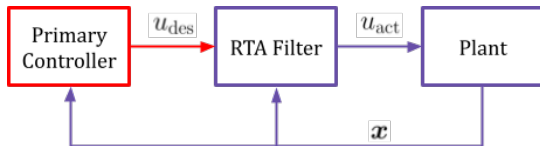


Figure 2. Feedback Control System with RTA

In this system, the primary controller first determines a desired control input u_{des} given the state x , which is passed to the RTA filter. If u_{des} is determined to be unsafe, the RTA filter will intervene and pass a safe action to the plant, referred to as u_{act} . If u_{des} is determined to be safe, the RTA

filter passes the action through unaltered. This control system structure allows the designer to isolate components with low safety confidence and assure that the entire system is safe. For this paper, the primary controller is the RL agent, the plant is the RL environment, and u_{des} is determined given the policy. By using RTA, the system could be designed such that the RL agent's only objective is to optimize performance, and RTA can be relied on to assure safety. However, in this first set of benchmarks, safety is primarily dealt with via reward shaping (penalizing unsafe actions).

The Aerospace SafeRL Framework was designed considering two types of RTA filters: Simplex based and Active Set Invariance Filter (ASIF) based approaches. For both approaches, a set of inequality safety constraint functions $h(x)$ are first defined, where the state x is considered to be safe if $h(x) \geq 0$. Simplex based approaches simply switch between a primary controller and backup controller [10]. In this case, the primary control input u_{des} is used if it is determined to be safe according to the safety constraints, and otherwise the backup control u_b is used. One possible implementation of a Simplex filter is as follows, where $\phi^{u_{\text{des}}}(x)$ is a prediction of the state x under the desired control u_{des} .

Simplex Filter

$$u_{\text{act}}(x) = \begin{cases} u_{\text{des}}(x) & \text{if } h(\phi^{u_{\text{des}}}(x)) \geq 0 \\ u_b(x) & \text{if otherwise} \end{cases} \quad (1)$$

ASIF-based approaches use an optimization algorithm to minimize deviation from the primary control signal while still assuring safety [11, 12]. The optimization objective maintains safety by inhibiting progression of the plant state space outside of the safe set given a set of M control invariant safety constraints $BC(x, u)$ [13], derived by applying Nagumo's condition [14] to the safe set (i.e. constraining the plant's state transition flow with respect to the gradient of the safety constraints at the boundary). One possible implementation of an ASIF filter is as follows.

Active Set Invariance Filter

$$u_{\text{act}}(x) = \underset{u}{\text{argmin}} \|u_{\text{des}} - u\|^2 \quad \text{s.t. } BC_i(x, u) \geq 0, \quad \forall i \in \{1, \dots, M\} \quad (2)$$

Assuming control affine dynamics, a continuous-time system model is given by,

$$\dot{x} = f(x) + g(x)u \quad (3)$$

To enforce Nagumo's condition around the boundary of the set created by $h(x)$, the barrier constraints can be defined as,

$$BC(x, u) := L_f h(x) + L_g h(x)u + \alpha(h(x)) \geq 0 \quad (4)$$

where L_f and L_g are Lie derivatives of f and g respectively, and α is a class κ strengthening function used to relax the constraints away from the boundary.

Considerations for Benchmark Comparison Metrics

Measuring the performance of the agent on a task and aggregating over multiple episodes can be used to gauge the effectiveness of an RL agent. Section 5 describes the various performance, efficiency, and safety metrics proposed for the

Aerospace SafeRL Framework. While the ultimate goal is to produce an agent with the best possible performance by the end of training, it is common to evaluate agent performance at different points during the training process as well as agent performance after training.

Training Curve Insights—Training curves, like those shown using TensorBoard, are a useful measure of *incremental performance* as well as *stability* of the RL solution. The most efficient way to quantify agent performance improvement while training is to leverage evaluation episodes generated during the training process. While evaluation episodes generally constitute a representative proxy for true policy performance in on-policy methods such as PPO, there are some potential drawbacks depending on RL algorithm, hyperparameters, and hardware resources [15]. For example, individual training iterations may utilize relatively low numbers of episodes if the train batch size is small or episodes are long, resulting in poor sampling and noise. Additionally, because iterations are defined by numbers of interactions, and not by number of episodes, a single episode may be split up across two or more iterations, obfuscating which iteration the agent behavior represents during evaluation. Ray RLlib utilized by Aerospace SafeRL supports metrics logging of both training episodes and evaluation episodes via TensorBoard and CSV. Performance of RL agents may exhibit instability and fragility during the training processes. This can come from random noise in episode initial condition sampling or large gradient updates to the agent model overshooting and degrading quality. Note that this process is not necessarily bad, it is often desirable to leave fragile local optimum to find a better, more stable global optimum. While PPO is more robust to these types of degradation, it is still beneficial to assess the stability of and average performance over many episodes to select the best model parameters checkpoint.

Addressing Stochastic Results—As alluded to earlier, initial conditions for episodes are drawn randomly. This ensures that during training and evaluation the agent encounters a variety of conditions, reflecting the unpredictability of real-world conditions. However, due to sampling noise, it is necessary to sample a large number of episodes during evaluation to ensure that good/bad performance isn't due to being lucky/unlucky. It is possible to use a predetermined set of evaluation initial conditions to promote fairness and test corner cases; however, it is vital that these test cases are not consulted during training or hyperparameter tuning to prevent overfitting. Additionally, when evaluating training curves, it is important to repeat the training process multiple times with 5-10 different random seeds to average out the same sampling noise that may cause one training run to work better than another [15].

3. ENVIRONMENTS

The Aerospace SafeRL benchmarks focus on two general problems: training an unmanned wingman to fly in formation with a manned flight lead, and training a active deputy spacecraft to dock with a passive chief spacecraft. Both environments use classic aerospace dynamics models: Dubins aircraft models [16], and Clohessy-Wiltshire relative motion spacecraft dynamics [17] in Hill's reference frame [18].

2D Aircraft Formation Flight

In the 2D aircraft formation flight environment, the state of the system is defined as $\mathbf{x} = [x_L, y_L, \psi_L, v_L, x_W, y_W, \psi_W, v_W]^T \in \mathcal{X} \subset \mathbb{R}^8$, and includes the position (x, y) , heading

ψ , and velocity v of the lead L and wingman W aircraft. The control for the system is defined by $\mathbf{u} = [\dot{\psi}_W, \dot{v}_W]^T = [u_{W1}, u_{W2}]^T \in \mathcal{U} \subset \mathbb{R}^2$.

Dynamics—A dynamics of both the wingman and lead aircraft are computed using Eq. (5).

$$\begin{aligned}\dot{x} &= v \cos \psi \\ \dot{y} &= v \sin \psi \\ \dot{\psi} &= u_1 \\ \dot{v} &= u_2\end{aligned}\tag{5}$$

Success Criteria—The goal of the RL agent is to control the wingman aircraft to achieve a relative position behind the flight lead, which flies with a constant heading rate and velocity. The RL agent must move the wingman aircraft to a position defined by a relative distance ρ_r and an aspect angle θ_{AA} measured from the back of the lead aircraft ($-x_L^b$) to the vector pointing to the wingman's location, as depicted in Figure 4. This relative position command can be converted

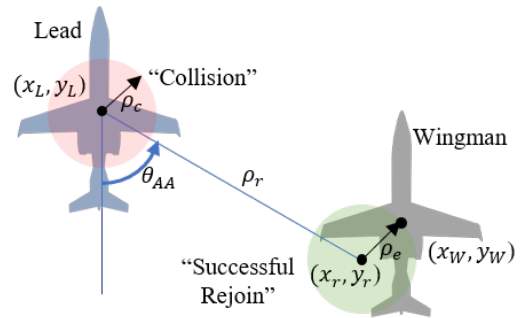


Figure 3. Depiction of general rejoin task

to Cartesian coordinates by Eq. 6.

$$\begin{aligned}x_r &= x_L + \rho_r \cos(\psi_L + \pi + \theta_{AA}) \\ y_r &= y_L + \rho_r \sin(\psi_L + \pi + \theta_{AA})\end{aligned}\tag{6}$$

An agent is considered successful if the error between the wingman's position and the commanded rejoin point is less than a specified value ρ_e and if the time in the rejoin position t_{rejoin} is greater than a threshold value $t_{success}$ (in this case 20 seconds).

$$\begin{aligned}\varphi_{\text{formation}} &: (\sqrt{(x_W - x_r)^2 + (y_W - y_r)^2} \leq \rho_e) \\ &\wedge (t_{rejoin} \geq t_{success})\end{aligned}\tag{7}$$

Safety Constraint—The RL agent must learn to rejoin while adhering to a collision avoidance safety constraint defined as maintaining a safe minimum distance ρ_c .

$$\varphi_{s_{2DAC}} := \sqrt{(x_L - x_W)^2 + (y_L - y_W)^2} \geq \rho_c\tag{8}$$

3D Aircraft Formation Flight

In the 3D formation flight environment, the state of the aircraft $\mathbf{x} = [x_L, y_L, z_L, \psi_L, v_L, \gamma_L, \phi_L, x_W, y_W, z_W, \psi_W, v_W, \gamma_W, \phi_W]^T \in \mathcal{X} \subset \mathbb{R}^{14}$ is the position (x, y, z) , heading ψ , velocity v , flight path angle γ , and roll angle ϕ of the lead L and wingman W aircraft. The control for the system is defined by $\mathbf{u} = [\dot{v}_L, \dot{\gamma}_L, \dot{\phi}_L, \dot{v}_W, \dot{\gamma}_W, \dot{\phi}_W]^T = [u_{L1}, u_{L2}, u_{L3}, u_{W1}, u_{W2}, u_{W3}]^T \in \mathcal{U} \subset \mathbb{R}^6$.

Dynamics—The dynamics of the each aircraft are defined by Eq. 9, originally presented in [19].

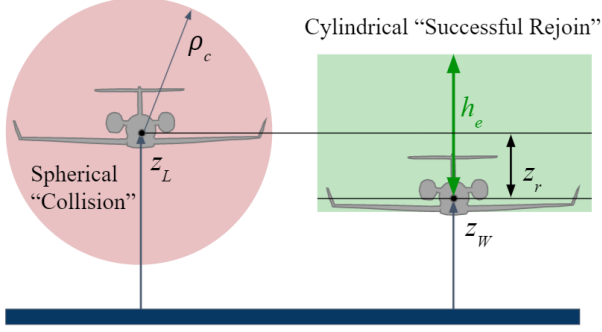


Figure 4. Depiction of general rejoin task z components

$$\begin{aligned}
 \dot{x} &= v \cos \psi \cos \gamma \\
 \dot{y} &= v \sin \psi \cos \gamma \\
 \dot{z} &= -v \sin \gamma \\
 \dot{\psi} &= \frac{g}{v} \tan \phi \\
 \dot{v} &= u_1 \\
 \dot{\gamma} &= u_2 \\
 \dot{\phi} &= u_3
 \end{aligned} \tag{9}$$

Success Criteria—In a slight modification of the 2D case, the 3D aircraft formation flight agent is considered successful when

$$\begin{aligned}
 \varphi_{\text{formation}} : & (\sqrt{(x_W - x_r)^2 + (y_W - y_r)^2} \leq \rho_e) \\
 & \wedge (|z_W - z_r| \leq \frac{h_e}{2}) \wedge (t_{\text{rejoin}} \geq t_{\text{success}}),
 \end{aligned} \tag{10}$$

where h_e is the height of the 3D rejoin cylinder centered on the rejoin point.

Safety Constraint—The RL agent must learn to rejoin while adhering to a collision avoidance safety constraint defined as maintaining a safe minimum distance ρ_c .

$$\begin{aligned}
 \varphi_{s3DAC} := & \sqrt{(x_L - x_W)^2 + (y_L - y_W)^2 + (z_L - z_W)^2} \\
 & \geq \rho_c
 \end{aligned} \tag{11}$$

2D Spacecraft Docking

In the 2D spacecraft docking environment, the state of an active deputy spacecraft is expressed relative to the passive chief spacecraft in Hill's reference frame [18] $\mathcal{F}_H := (\mathcal{O}_H, \hat{i}_H, \hat{j}_H)$. The origin of Hill's frame \mathcal{O}_H is located at the mass center of the chief, the unit vector \hat{i}_H points away from the Earth along a line connecting the center of Earth to \mathcal{O}_H , and the unit vector \hat{j}_H is aligned with the orbital velocity vector of the chief. The state of the deputy is defined as $\mathbf{x} = [x, y, \dot{x}, \dot{y}]^T \in \mathcal{X} \subset \mathbb{R}^4$, where, $\mathbf{r} = x\hat{i}_H + y\hat{j}_H$ is the position vector and $\mathbf{v} = \dot{x}\hat{i}_H + \dot{y}\hat{j}_H$ is the velocity vector of the deputy in Hills Frame. The control for the system is defined by $\mathbf{u} = [F_x, F_y]^T = [u_1, u_2]^T \in \mathcal{U} \subset \mathbb{R}^2$.

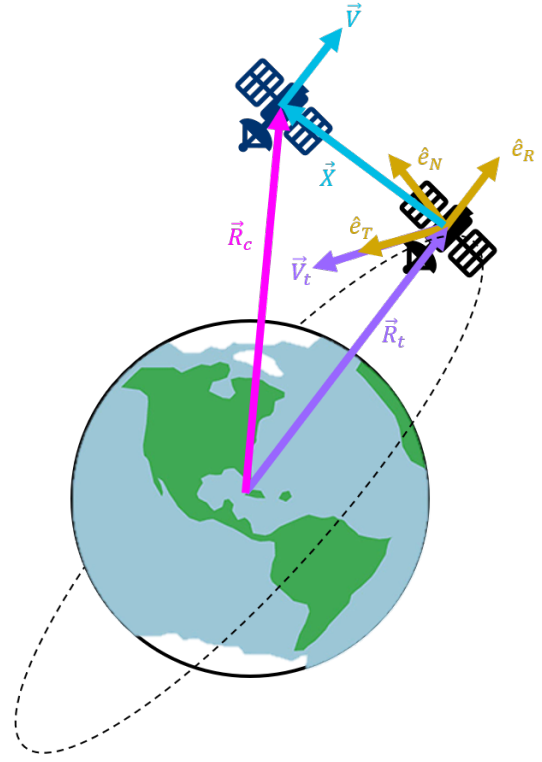


Figure 5. Hill's reference frame centered on a chief spacecraft and used to describe the relative motion of a deputy spacecraft conducting proximity operations (not to scale).

Dynamics—A first order approximation of the relative motion dynamics between the deputy and chief spacecraft is given by Clohessy-Wiltshire [17] equations,

$$\begin{aligned}
 \ddot{x} &= 2n\dot{y} + 3n^2x + \frac{F_x}{m} \\
 \ddot{y} &= -2n\dot{x} + \frac{F_y}{m}
 \end{aligned} \tag{12}$$

where n is spacecraft mean motion and m is the mass of the deputy.

Success Criteria—The deputy is considered successfully docked when its distance to the chief is less than a desired distance ρ_d .

$$\varphi_{\text{docking}} : (\|\mathbf{r}_H\| \leq \rho_d) \tag{13}$$

Safety Constraint—The RL agent must learn to dock while adhering to a dynamic velocity safety constraint that restricts the relative velocity of the deputy to velocity limit that decreases as it approaches the chief. The system is defined to be safe if it obeys the following safety constraint for all time,

$$\varphi_{s2DSC} := \|\mathbf{v}_H\| \leq \nu_0 + \nu_1 \|\mathbf{r}_H\| \tag{14}$$

where, $\nu_0, \nu_1 \in \mathbb{R}_{\geq 0}$, and

$$\|\mathbf{r}_H\| = (x^2 + y^2)^{1/2}, \quad \|\mathbf{v}_H\| = (\dot{x}^2 + \dot{y}^2)^{1/2}. \tag{15}$$

The constraint in Eq. (17) enacts a distance-dependent speed limit, with ν_0 defining the maximum allowable docking speed and ν_1 defining the rate at which deputy must slow down as

it approaches the chief. The values $\nu_0 = 0.2$ m/s, and $\nu_1 = 2n$ s⁻¹ are selected based on elliptical closed natural motion trajectories (eCNMT), and further insight into this choice is given in [20, 21].

3D Spacecraft Docking

The 3D spacecraft docking problem builds on the 2D problem by adding the z coordinates. Since the in-plane (x - y) and out-of-plane (z) Clohessy-Wiltshire dynamics are decoupled, this problem simply adds a z -dimension. The state of the deputy is defined as $\mathbf{x} = [x, y, z, \dot{x}, \dot{y}, \dot{z}]^T \in \mathcal{X} \subset \mathbb{R}^6$, and the control for the system is defined by $\mathbf{u} = [F_x, F_y, F_z]^T = [u_1, u_2, u_3]^T \in \mathcal{U} \subset \mathbb{R}^3$.

Dynamics—The dynamics in the z direction are given by

$$\ddot{z} = -n^2 z + \frac{F_z}{m}. \quad (16)$$

where n is spacecraft mean motion and m is the mass of the deputy.

Success Criteria—The deputy is considered successfully docked when its distance to the chief is less than a desired distance ρ_d , as defined in Eq. 13, where $\|\mathbf{r}_H\| = (x^2 + y^2 + z^2)^{1/2}$ in the 3D version.

Safety Constraint—The safety constraint also features a slight modification:

$$\varphi_{s3DSC} := \|\mathbf{v}_H\| \leq \nu_0 + \nu_1 \|\mathbf{r}_H\| \quad (17)$$

where, $\nu_0, \nu_1 \in \mathbb{R}_{\geq 0}$, and $\|\mathbf{v}_H\| = (\dot{x}^2 + \dot{y}^2 + \dot{z}^2)^{1/2}$.

2D Oriented Docking

This is a modification of the 2D docking problem to incorporate spacecraft orientation. The full description of this problem is available in a previous publication [22] and a summary is provided here for clarity.

Dynamics—In this model, thrust is only possible from two thrusters assumed to be perfectly aligned with the spacecraft body x -axis x_b on either side of the spacecraft. The attitude of the spacecraft about its z -axis n_{rw} is controlled by a reaction wheel and the rotational equations of motion are dictated by the conservation of angular momentum. As the inertia of the spacecraft about that axis is I_{zz} , the inertia of a reaction wheel about that spin axis then D , and θ_3 is the rotation of the spacecraft about \hat{e}_N , measured from the \hat{e}_T direction to the x_b axis, the rotational equations of motion are

$$I_{zz}\ddot{\theta}_3 = -D\dot{\psi} \quad (18)$$

where $\dot{\psi}$ is commanded acceleration given to the reaction wheel to produce a required torque. The state of the deputy is defined as $\mathbf{x} = [x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]^T \in \mathcal{X} \subset \mathbb{R}^6$, and the control for the system is defined by $\mathbf{u} = [F_x, \dot{\psi}]^T = [u_1, u_2]^T \in \mathcal{U} \subset \mathbb{R}^2$.

The 2D docking equations are then modified to include the the angular component and the the force of thrust in the x direction only F_x . Then

$$\vec{F}_{xy} = \begin{bmatrix} \cos(\theta_3) \\ \sin(\theta_3) \end{bmatrix} F_x. \quad (19)$$

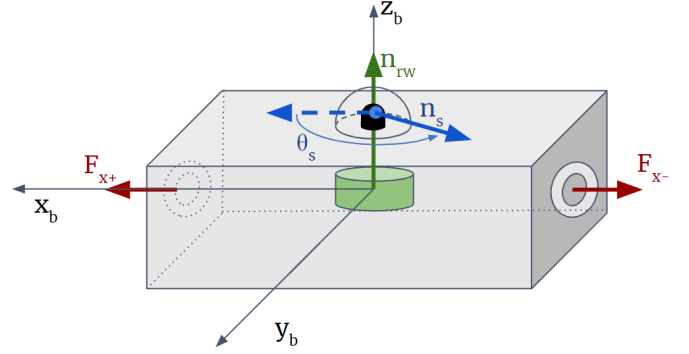


Figure 6. Hypothetical 6U Cubesat with thrusters (red) aligned with positive and negative x -axes, a sensor (blue) that is able to gimbal around the z -axis to any direction within the xy -plane, and a reaction wheel (green) for single axis attitude control aligned with the z -axis [22].

Then, (12) can be rewritten as

$$\begin{aligned} \ddot{x} &= 2n\dot{y} + 3n^2x + \frac{F_x \cos(\theta_3)}{m} \\ \ddot{y} &= -2n\dot{x} + \frac{F_x \sin(\theta_3)}{m} \\ \ddot{\theta}_3 &= -\frac{D}{I_{zz}}\dot{\psi} \end{aligned} \quad (20)$$

Success Criteria and Safety Constraints—The success criteria and safety constraints for the 2D oriented are the same as the regular 2D case, defined by Eqns. 13 and 14, respectively.

4. AEROSPACE SAFERL ARCHITECTURE

The architecture of the Aerospace SafeRL Framework was designed with modularity in mind. This section describes the functional components of the architecture.

Framework Architecture

The Aerospace SafeRL framework is designed to use neural network agents and RL training algorithms from an external library, such as Ray RLlib [5], as shown in Figure 7. By utilizing these external libraries, Aerospace SafeRL gains access to mature RL algorithm implementations. While the Aerospace SafeRL Framework works best with RLlib, Environment interaction complies with the standard OpenAI Gym API [3] enabling any compatible RL implementation to be used instead.

The primary contribution of the Aerospace SafeRL Framework is contained within the Environment and Evaluation modules. The *Environment* module, shown in green in Figure 7, provides an interactive simulation environment to receive actions from the agent, simulate the result of taking those actions, and return resulting observations and rewards. This interaction is accomplished via an OpenAI Gym-compliant interface where the following quantities are exchanged between the agent and the environment.

Obs—The observation vector of the current state of the environment is passed from the environment to the agent. The agent perceives the environment through this observation vector, which can be thought of as the output of sensors in the environment. The observation may be the result of an

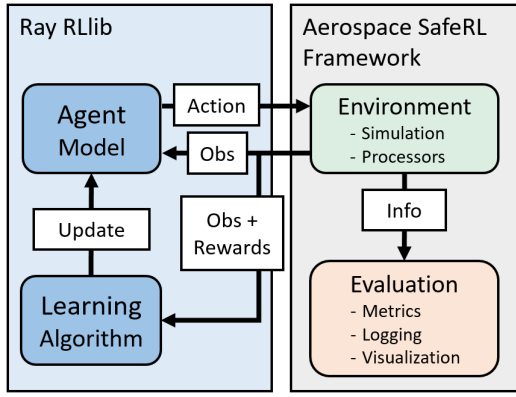


Figure 7. Aerospace SafeRL Framework Architecture.

environment which is fully-observable, i.e. Obs contains full knowledge of the entire state of the environment, or partially-observable, i.e. Obs contains incomplete information about the state of the environment. Obs may also undergo transformations to make it more amenable to processing by a neural network model. Dimensions and ranges are defined by the environment’s observation space

Action—The desired action vector is passed from the agent to the environment as defined by the agent’s policy $\pi(o)$. The Aerospace SafeRL Framework supports both discrete and continuous action spaces. Valid dimensions and ranges for actions are defined by the environment’s action space.

Reward—The reward is a scalar quantity grading the “goodness” of the agent’s action computed by the environment and used by the RL training algorithm to improve the agent’s policy. Rewards may be dense, providing frequent, short-term feedback on actions taken, or sparse, providing infrequent, long-term feedback on actions taken. For example, dense rewards may provide small rewards for getting closer to a goal, whereas sparse rewards may only offer a single large reward once a goal is actually reached. Rewards are implemented as the sum of many contributing component rewards.

Info—The information dictionary contains metadata about the state of the environment. This dictionary is ignored during RL, but used by Aerospace SafeRL for logging, debugging, and evaluation.

Done—The done variable is a simple Boolean value returned by the environment indicating that a terminal state has been reached and the episode is concluded. The done condition could be the result of multiple success or failure conditions.

Evaluation—The evaluation module contains utilities and scripts to evaluate agent performance and facilitate convenient usage of the framework. The evaluation module includes a set of SafeRL Metrics, described in Section 5, to standardize the performance and safety evaluation of agents on Aerospace SafeRL tasks. Additional scripts are provided to evaluate trained agent performance, visualize/animate task episodes, and generate result plots. Finally, logging of training episodes, via custom RLLib callbacks, enables application of these evaluation and visualization tools to increase understanding and interpretability of the RL training process.

Environment Architecture

The Environment module is responsible for simulating the environment, determining the observation of the environment, enforcing safety constraints, and providing reward feedback to the agent on its actions. Building RL environments typically requires significant engineering resources and presents a prohibitive barrier to entry for new applications and research. The deliberate modularity and consistent bookkeeping of the Environment module allows users of this framework to efficiently create new experiments and analysis.

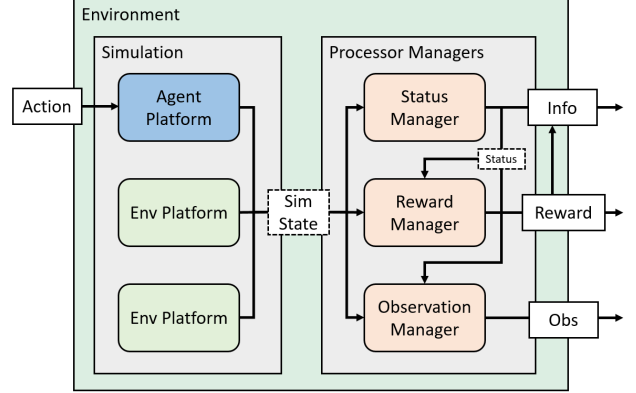


Figure 8. The Aerospace SafeRL Environment Architecture.

The Environment module is subdivided into the simulation and processors, as depicted in Figure 8. This split isolates the two roles of an RL environment: modeling the response of the world’s state to an agent’s actions and computing how the new world state affects the agent. The simulation propagates the dynamics and interactions of the individual platforms, e.g. lead and wingman aircraft, that compose the environment. The platforms include the RL agent as well as other agents in the environment, e.g. the wingman that is controlled by the RL agent, and the lead that is controlled by an external model (in this case, scripted). From the simulated state, the collective state of the simulations constituent platforms, the processors compute types of metadata from state such as status properties, agent rewards, and agent observations. Processors are of the same type are bundled together into managers where their outputs can be collected into the Obs, Reward, Info, and Done outputs of the environment.

Environment Configuration

Aerospace SafeRL is designed for modularity and customizability. To that end, the Environment module is composed entirely of modular, interchangeable components with fully generic glue code. All components of an environment, including the simulation platforms and processors, are specified in a yaml config file. The environment specific configuration is found under the environment config portion of the yaml config file. Just like the rest of the framework, the environment config has a modular structure where each individual component of the environment has an individual, self-contained configuration dictionary. There are four primary sections in the environment config: *environment objects* containing a list of individual environment object/platform configs, *status* containing a list of individual status processors configs, *reward* containing a list of individual reward processors configs, and *observation* containing a list of individual observation processors configs. Each individual subconfig is independently parsed into constructor parameters for the

specified object. The individual constructed objects are collected into their respective parent submodule, Simulation for environment objects/platforms and managers for processors, to create the full environment.

Simulation

As described above, the simulation is composed of individual *environment objects*. There are currently two types of environment objects: physical entity *platforms* like aircraft or spacecraft with actuators and state dynamics models, and *geometry* that describes abstract regions in space such as keep-out zones or goal areas. For example, in the rejoin environment, the wingman is an agent platform and the lead is an environment platform, while the rejoin region and collision zone centered on the lead are both geometry objects. At each timestep of the simulation, each individual environment object performs an individual state transition step and increments the collective simulation state. This new simulation state is then passed to the processors to produce the environment step output. All environment objects are interchangeable and implement the following set of standard properties: name, position, velocity, and orientation.

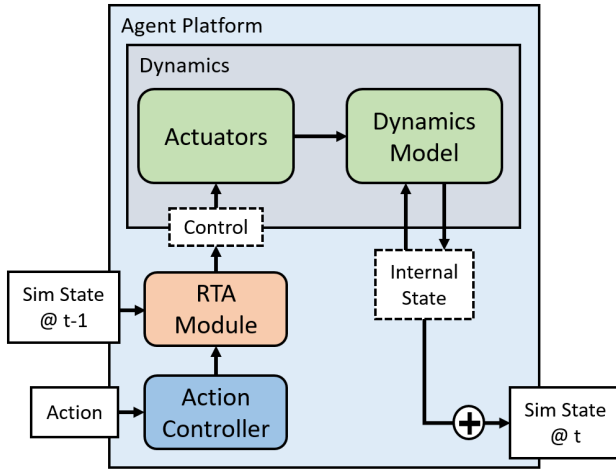


Figure 9. Aerospace SafeRL Simulation Platform Architecture

Platforms—Each platform maintains an internal state object, potentially vectorized, to simplify atomic copying and future trajectory prediction. The next state transition is determined by the platform’s dynamics model which computes the future state based on the current state and control vector. The control vector input to the dynamics model is produced through the successive chain of controller, RTA, and actuator. In this case, actuator is meant generally and can describe inner loop controls such as control surface deflection, outerloop actuators, or more abstract forms of control such as forces and torques. The controller produces a desired per-actuator control in response to the current simulation state.

Run Time Assurance Module—The RTA Module within an individual platform provides online safety assurance in the manner described in Section 2. The Aerospace SafeRL Framework provides a standardized interface and platform initializer hooks to allow simple implementation and integration of custom RTA solutions. The generic RTA module interface simply consists of a *filter control* method that produces a safe control vector given a desired control input and current simulation state and an *intervening* flag indicating when the RTA filter is actively modifying the desired control.

Extending this simple RTA behavior are Simplex based and ASIF based implementations. The Simplex RTA interface splits the control filtering into a *monitor* stage that checks for safety constraint violation and a known safe *backup control* that takes over when the safety constraint monitor triggers, as described by Eq. 1. Control is returned to the agent once relative distance from the safety constraints is achieved. The ASIF RTA generates an optimally small control modification given a set of safety constraints as described by Eq. 2. As developing this type of RTA can be challenging, Aerospace SafeRL contains a base ASIF RTA module for easy implementation and integration. For systems with linear dynamics, custom ASIF RTA can be specified with the ASIF RTA module interface by simply defining control invariant safety constraints from which the optimization problem is automatically derived and solved. Constraints are defined with RTA constraint objects and must implement an inequality constraint function $h(x)$, constraint gradient $\nabla h(x)$, and a softening function $\alpha(x)$. With these required interface methods, the ASIF RTA module constructs barrier constraints from the safety constraints and solves a quadratic program to minimize the l^2 norm difference between the safe control and the desired control. If control invariant safety constraints cannot be defined, a backup controller can also be implemented such that optimization occurs over a backup trajectory rather than at a single point.

Initialization—In order for the RL agent to learn a good policy, it is necessary for it to interact with a representative sample of possible state spaces. Simulation initialization at the beginning of each episode is vital to both define the bounds of the task and to provide that varied sampling. The initializer is specified in the config and may either be a generic random bounds initializer, which draws property values from specified random bounds, or a custom initializer class implemented in python. The framework also supports initialization dependency, where the output of an environment object’s initializer depends on the output of another environment object. For example, the wingman aircraft is initialized in a random point within a minimum and maximum distance relative to the lead aircraft.

Processors

After the simulation computes the simulation state at the current timestep, feedback must be provided to the agent. Status, rewards, and observations are computed by their respective processor managers shown on the right side of Figure 8. The managers depicted in Figure 10 encapsulate the individual processors and aggregate their outputs into the final quantity while collecting useful metadata that can be added to logging outputs, such as the contribution of each reward processor to the total reward both per step and per episode. Managers also enforce processor execution order allowing later processors to depend on the output of previous ones. Additional status, rewards, or observations or observations can be added by implementing a new processor and/or specifying it in the config file.

Processor State Machine—Individual processors are modeled as state machines that maintain an internal state and use this state to generate their output. As shown in Figure 11, at each timestep of length τ , the status, reward, and observation managers transition their processors’ internal states in response to the new simulation state. Status comes first, allowing reward and obs processors to access status values during their state transitions and output computations. The state transitions of the internal processor states, X_i^S , X_i^R , X_i^O , and their outputs,

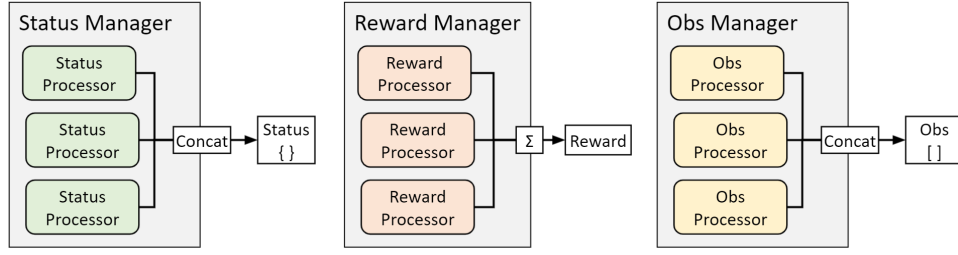


Figure 10. Processor Managers

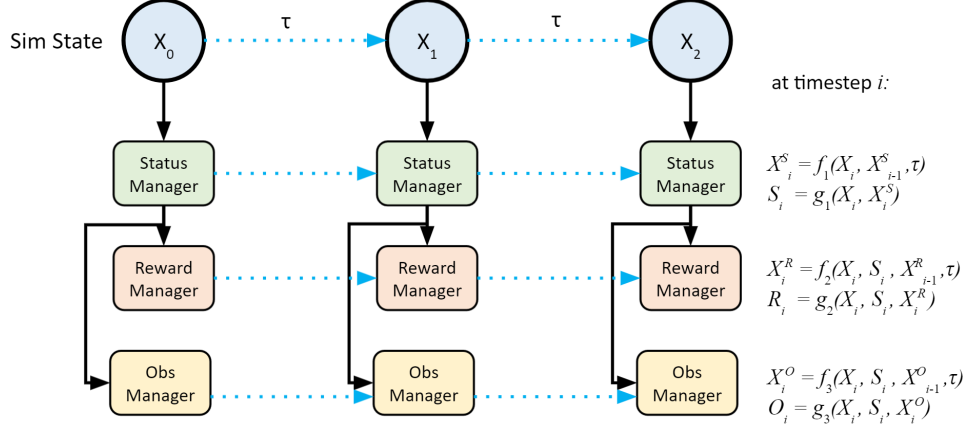


Figure 11. Processor/Manager State Machine

S_i, R_i, O_i , are shown in terms of functions on the right side of the figure where X_i corresponds to the simulation state at time $t_i = i\tau$.

Processor Interface—The interface of a processor consists of two methods, `increment()` which propagates the internal state to the and `process()` which computes the processor’s output. Both methods are functions of the the simulation state and the processor’s internal state, with an additional input of the timestep length for `increment()`. Limiting processor dependence on only these quantities enforces processor modularity allowing them to be inserted/removed/modified freely. Processors may have acyclic dependencies on the output of other status processors whose status’ are added to the simulation state as they are called in sequence. When specifying a processor list with such dependencies in the config, every processor must come after its parents in the acyclic dependency graph.

Implementing these methods to manage the processor’s state transition properly is vital. Importantly, `process()` should never modify the internal state of the processor and can therefore be called without modifying the environment’s Markov state. This is particularly important for environment initialization. On the other hand, `increment()` does modify the environment state and is therefore called exactly once per environment step. For a simple example, a status processor that keeps track of the amount of time spent in a goal area may increment an internal timer in `increment()` and output the contents of the stored timer in `process()`. Note that not all processors require an internal state and therefore do not have to implement `increment()`. While it takes a bit more effort, decomposing the processors into these two separate methods ensures clean software design and broader compatibility.

5. METRICS

This section defines metrics to compare different reinforcement learning approaches, different approaches to training with safety, different safety bounding approaches, and different neural network verification metrics.

The first area of evaluation is comparing agent performance under different neural network architectures, RL algorithms, rewards, use of RTA, and hyperparameter combinations. The following metrics are given to evaluate the performance of different neural networks on the aerospace benchmarks.

- **Average Return (*higher is better*)** - The average cumulative reward earned by agent during an episode. This is the standard performance metric for evaluating RL agents and is the most direct proxy for agent performance as maximizing this quantity is the explicit goal of RL. Note that this is not necessarily the be-all, end-all indicator for agent quality as reward is only a proxy for agent performance used to guide the optimization of the policy and may be modified to that end. Additionally, RL agents typically maximize a future-discounted cumulative reward further amplifying this effect.
- **Episode Length (*lower is better*)** - The average episode length across all episodes in the final iteration of training. In the case that a task has a duration associated with it, the time to complete the task is measured as the time to complete the minimum requirement. For example, if the task is that the autonomous satellite should dock with another satellite or that a wingman aircraft should move to a specified formation flight position with a lead aircraft, task completion time is the time to achieve the combination of position and velocity requirements defining docked or in formation. For the rejoin and docking environments, this is measured with mean episode length.
- **Success Rate (*higher is better*)** - The percentage of episodes in the final iteration of training that successfully

completed the goal. Ideally the last iteration will be 100% successful; however, this is not always the case. For example, this might be the percent of cases that successfully dock a deputy to a chief spacecraft, or successfully place a wingman aircraft in formation flight with a flight lead aircraft.

- **Interaction Efficiency Rate (*lower is better*)** - The number of interactions (i.e. samples, or timesteps) required to train to some level of performance. This measures how quickly the agent learns to perform the task. In this work, we choose to evaluate a rejoin and docking performance on an 80% success mean threshold, although any criteria could be chosen.

- **Control Usage (*lower is better*)** - The amount of fuel, cumulative control, and/or battery life used to successfully complete the task. For example, in a spacecraft environment, this might be measured as the total change in velocity over the task. This is not implemented in the rejoin environment.

In SafeRL, a key set of metrics concern whether the agent violates safety constraints during or after training. Example metrics to measure safety are as follows:

- **Safety Violation Rate (*lower is better*)** - The percent of test cases in which the control system did not at any time violate safety constraints. This could be measured as a boolean value, where if safety is violated at any time, that test or training case is unsafe. For example, if the docking spacecraft violates the velocity limit at any point or the wingman aircraft violates safe separation distance, then the training or test case would be considered unsafe.

- **Episode Safety Violation Rate (*lower is better*)** - The percentage of time that the agent command is unsafe for an episode. Rather than looking at the problem from a boolean safe/unsafe episode level, this metric looks at the severity of the violation in terms of time. For example, this may look at the number of timesteps that the docking simulation violated the velocity constraint or the number of timesteps that the formation flight simulation violated the safe separation constraint.

- **Episode Violation Severity (*lower is better*)** - A measure of how unsafe the control action was. For example in the formation flight case, if a wingman violates a safe separation distance with the lead, this metric could measure how far the wingman entered the “collision” sphere as a percentage of the total unsafe distance. In the case of spacecraft docking this metric may measure how much the spacecraft violated the velocity constraint as a percent of the velocity constraint value.

In addition to measuring safety for scenarios it can be valuable to look at **cases outside the original training set**. For example, if training cases only included docking from up to 1 kilometer away, a series of test cases may evaluate docking from 10 kilometers away, to see how well adapted the neural network is to performing outside of it’s training set. A second set of the four RL metrics and three safety metrics could be included to specifically look at performance outside of the training region.

Two final metrics categories could be used under special conditions. In the case of *standardized rewards*, where evaluating different rewards is not a part of the exploration, **Mean Reward (*higher is better*)** can be used to compare approaches. In the case where training is completed consistently on a standard set of hardware, a **Training Computation Time (*lower is better*)** may be used to evaluate how long it takes to reach a fixed number of training interactions (timesteps).

6. BASELINE PERFORMANCE

This section presents a subset of the results of training and evaluating RL agent performance on these benchmark tasks. Full results may be generated by using the default configuration files for each environment. The purpose of these results is to establish an initial performance baseline from which further improvements will be made by the research community. The plots below include the total reward, task success rate, and average episode length the agents achieved while training on these environments. All results shown were produced with shallow feed-forward multilayer perceptron neural networks trained with Ray RLlib’s PPO implementation.

Discussion of training results

Figures 12 - 16 depict the results of training an agent on the 2D/3D Rejoin, 2D/3D Docking, and 2D Oriented Docking environments. Each plot is constructed with 10 different training runs, each with a different random seed. The shaded in regions in the graphs depicts the 95% confidence interval at that training environment interaction timestep over the 10 training runs. The plots generated correspond to the metrics listed in Section 5. For the Rejoin environments, three graphs each depict the training of the agent: success rate, episode length, and the average return. For the Docking environments, two additional graphs describe how often the velocity constraint described by Eq. 14 is violated as a proportion of the entire training episode as well as the average episode’s thrust induced Δv , which corresponds to fuel usage. As seen in the graphs, the 2D/3D Rejoin and 2D/3D Docking environments all converge near 100% success rate. The 2D Oriented Docking problem converges, albeit with less stability, to a respectable 90% success rate with some seeds reaching reliable 99% success rates and other seeds resulting in less effective policies. The episode length and average return graphs also converge to a minimum and maximum value, respectively. In all three Docking environments, while the average return is leveling out, the average episode length and Δv continue to gradually decline, indicating the agent is learning to dock more quickly and efficiently. This is shown very clearly in Figure 19. The relative ratio of timesteps with the velocity constraint violated, shown in Figures 13d, 15d, and 16d show the policies quickly learning to avoid the velocity constraint boundary and maintaining a low, although non-zero, degree of constraint violation. There is also an observable constraint violation ratio pattern, most evident in Figure 15d, where constraint violation increases to reach the initial success plateau around $1e6$ timesteps and then decays as the policy refines towards better constraint compliance and time efficiency.

The trajectories of the 2D rejoin agent at various points in the training process are presented in Figure 17, where the black dotted line represents the path of the flight lead, and the colored lines represent the wingman agent with increasing amounts of training interaction experience. The terminal points of each path represent where the wingman successfully rejoins. From the results, it can be seen that the wingman aircraft learns to rejoin earlier over more training and converges near an optimal trajectory. Similarly, Figure 18 shows the trajectory of a deputy spacecraft docking with the chief at the origin. Earlier training sessions take more indirect, circuitous paths, but as the training progresses, the agent learns to take a more optimal route towards the chief. Finally, Figure 19 shows the deputy’s velocity versus it’s distance to the chief in the 2D docking task with the velocity limit constraint superimposed as a black dotted line (safe is below the line). Early in the training, the velocity frequently exceeds

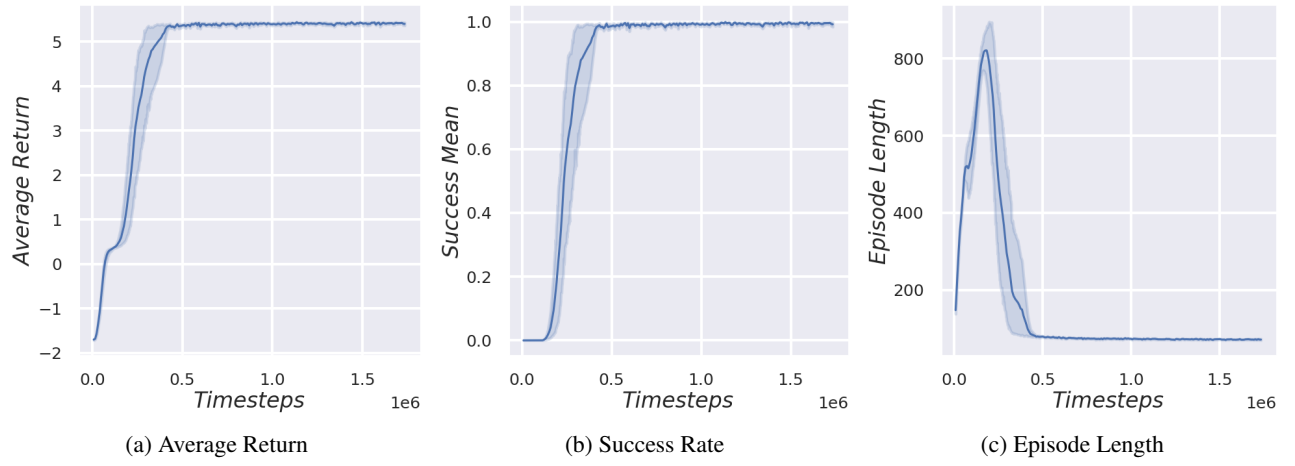


Figure 12. 2D Rejoin Training Performance Metrics

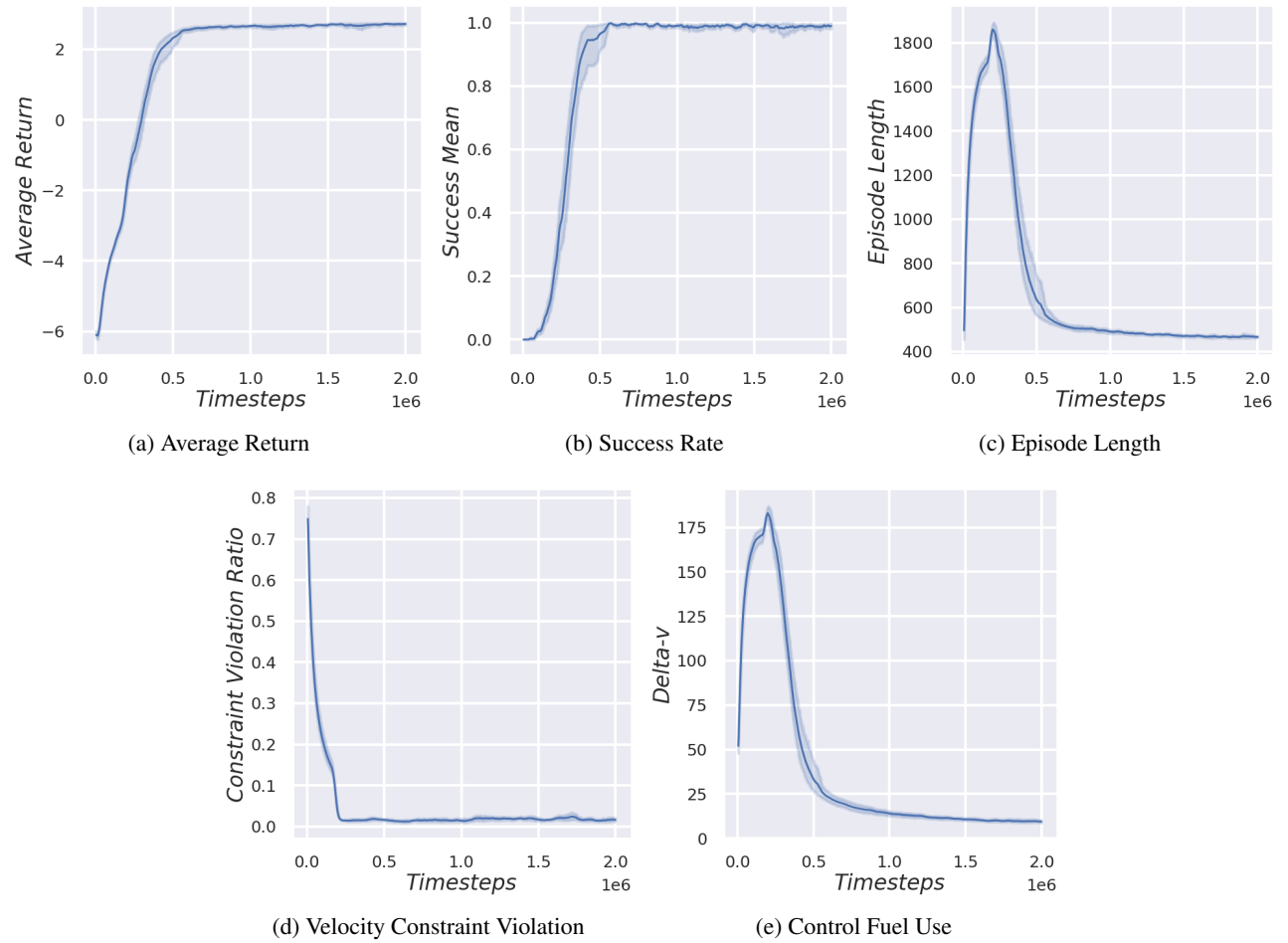


Figure 13. 2D Docking Training Performance Metrics

the distance-dependent limit while simultaneously moving to lower velocities in an overly-conservative effort to avoid the velocity constraints. Later in the training processes the docking policy manages to both avoid the velocity constraint more effectively while maintaining a more optimal velocity closer to the constraint bound. In addition, the policy evolves to waste less fuel oscillating between high and low velocities, matching the improved time efficiency with improved fuel

efficiency.

Table 2 shows the interaction efficiency of our baseline solution for each of the 5 environments. This quantity indicates the number of timesteps the agents had to interact with the environment before reaching an average success threshold of 80%. Unsurprisingly, the simpler the environment, the more efficiently the agents learn a successful policy.

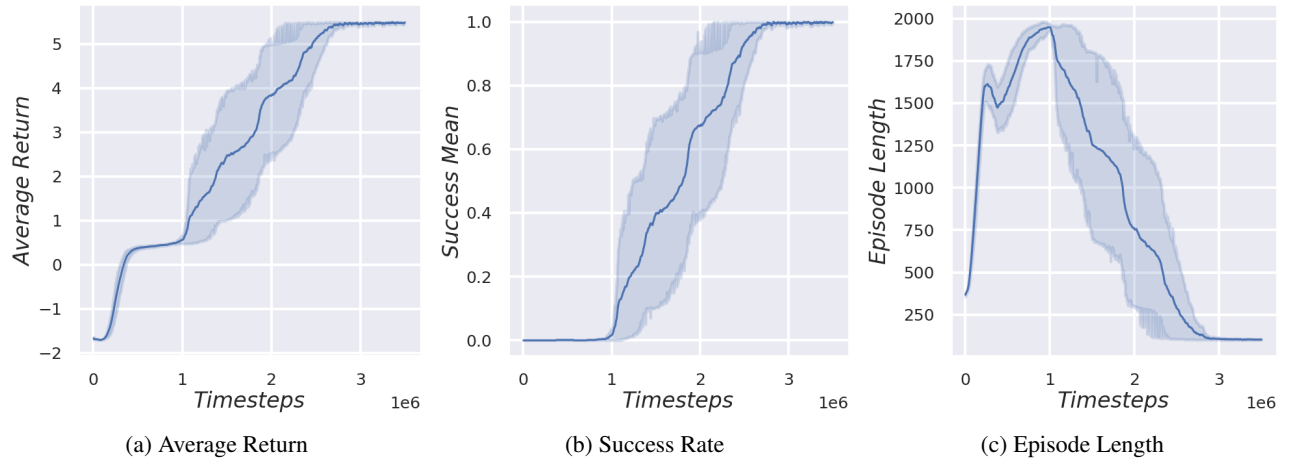


Figure 14. 3D Rejoin Training Performance Metrics

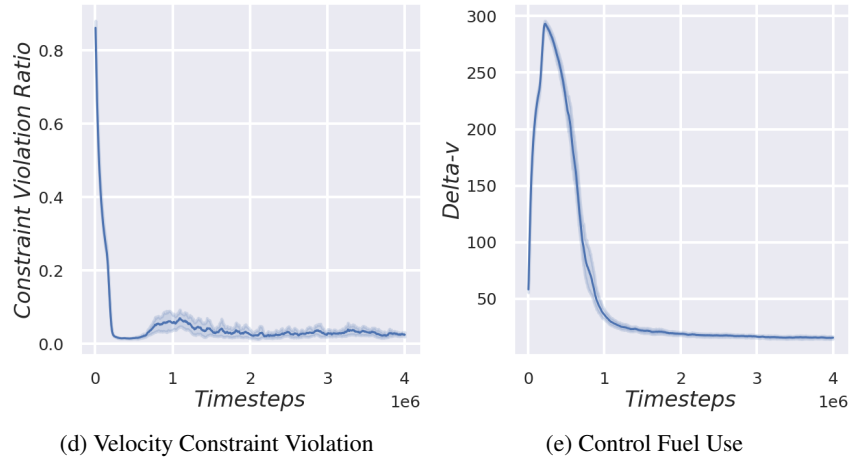
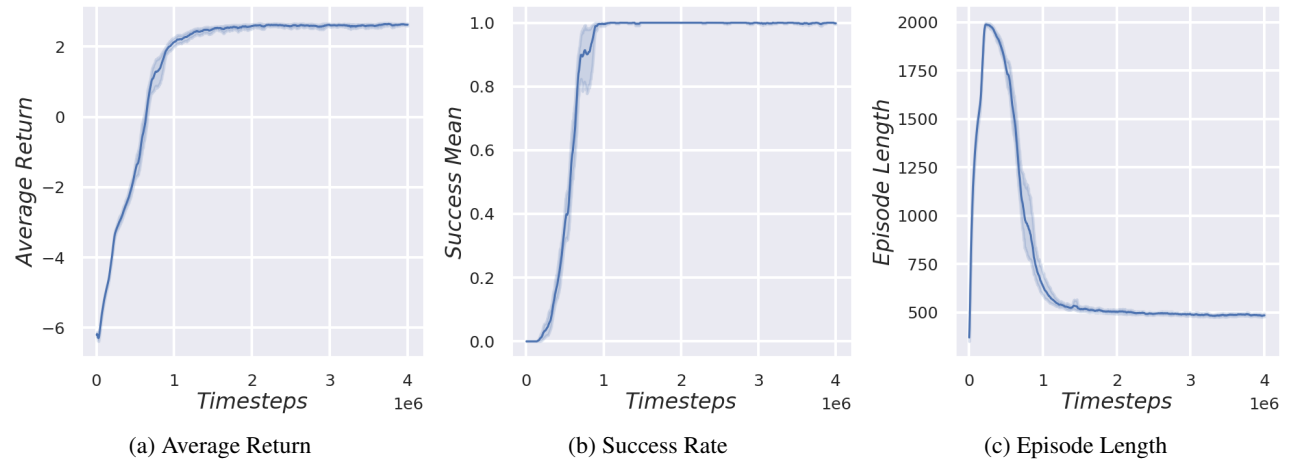


Figure 15. 3D Docking Training Performance Metrics

7. CONCLUSIONS AND RECOMMENDATIONS

The Aerospace SafeRL Framework fills a gap in standard aerospace application benchmarks of SafeRL. Five standard aerospace control benchmarks at varying levels of complexity (numbers of states, linear vs. nonlinear dynamics, etc.) were created with a default configuration to train an RL agent for each. These benchmarks serve as a foundation to investigate many open SafeRL research questions as well as a

mechanism to compare the impact of different RL algorithms, hyperparameter configurations, rewards, and RTA approaches on aerospace control applications. Future extensions of this work include adding additional aerospace environments by building on the modular architecture, investigating alternative neural network architectures beyond multilayer perceptrons, investigating the impact of novel RTA designs on the RL training process, developing alternative RL algorithms

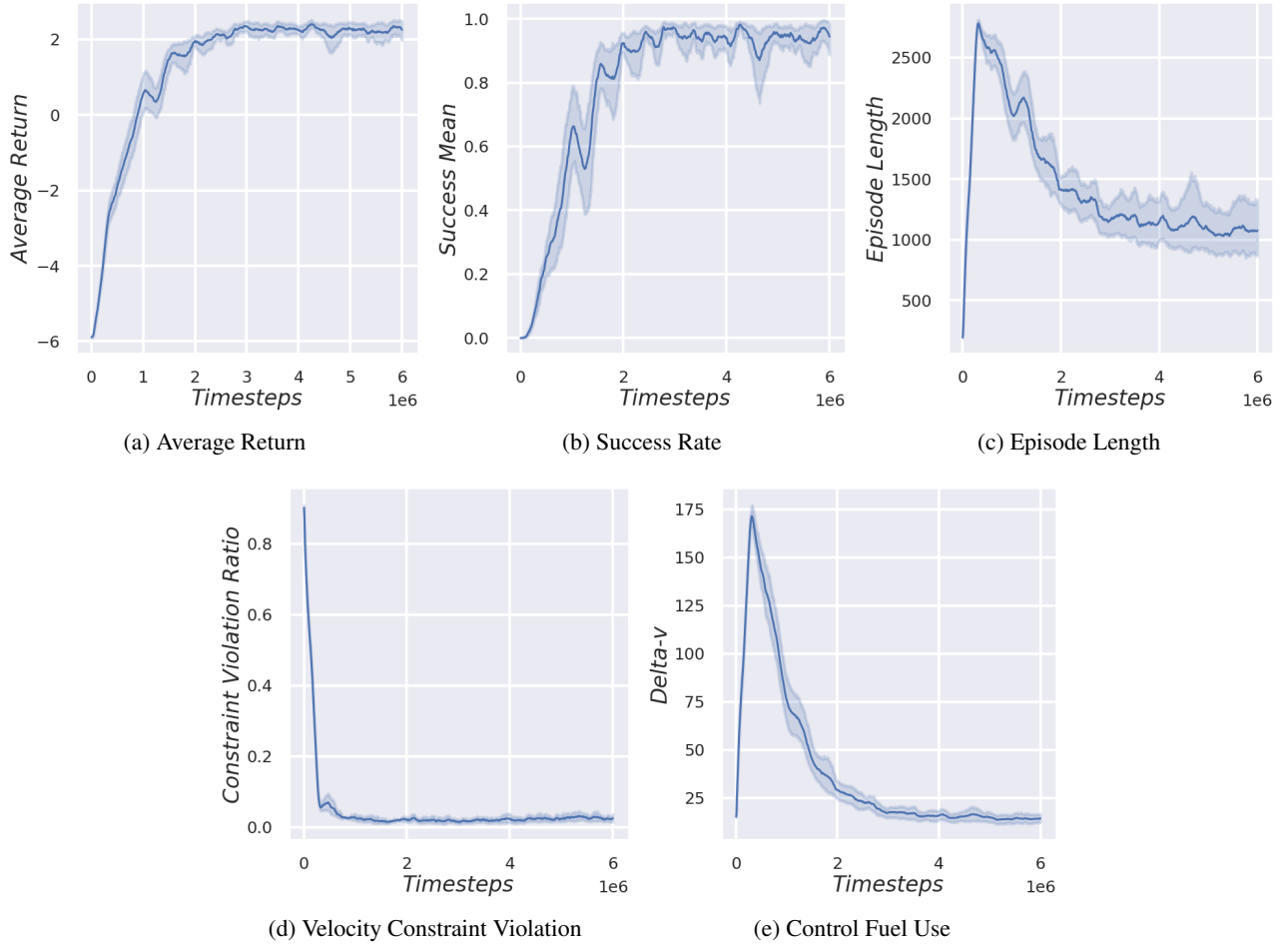


Figure 16. 2D Oriented Docking Training Performance Metrics

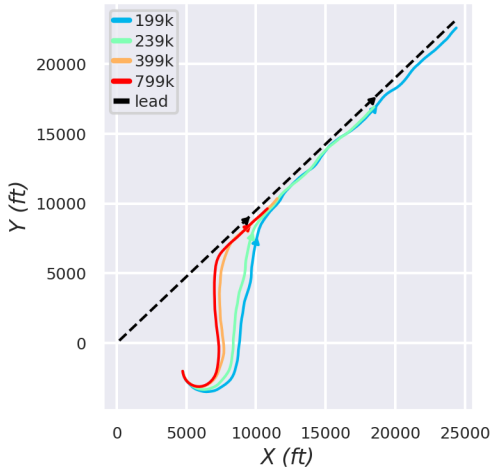


Figure 17. 2D Rejoin episodes during agent training progression.

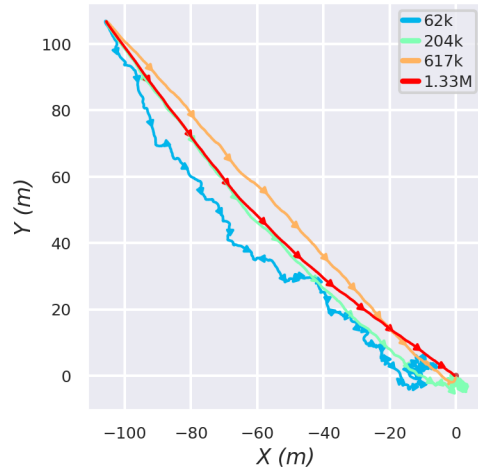


Figure 18. 2D Docking episodes during agent training progression.

with safety awareness, and applying neural network verification to find areas to retrain as part of the RL process. The benchmarks are available at <https://github.com/act3-ace/SafeRL>.

APPENDIX

This appendix provides additional details on the specific implementations of the five environments.

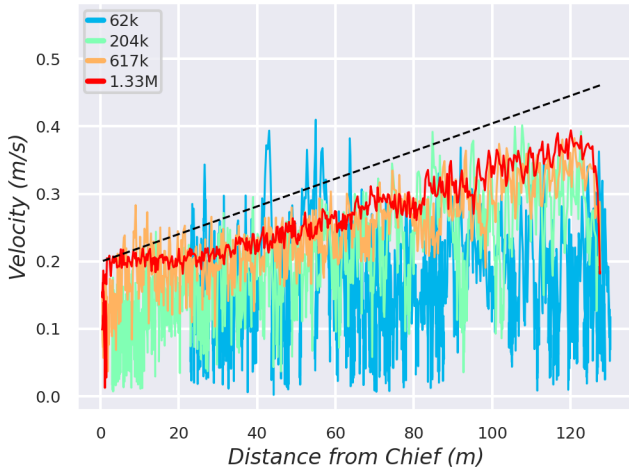


Figure 19. 2D Docking Velocity compared to Distanced-Based Velocity Limit.

Table 2. 80% Success Interaction Efficiency

Environment	Timesteps
2D Rejoin	0.30e6
3D Rejoin	2.3e6
2D Docking	0.34e6
3D Docking	0.66e6
2D Oriented Docking	1.5e6

Configuration File

The Aerospace SafeRL Framework’s modular architecture facilitates editing a single configuration file to change the task, environment platforms and their characteristics, the observation space, rewards, and status. In some cases, the classes in the framework defining components are listed, in others, specific values are assigned such as initial condition ranges, actuator limits, individual reward values.

Environment Platforms

The environment platforms are defined within the config file in an environment configuration. Each environment platform is given a name, a step size, state limitations, a controller specification, and a range of initial conditions for each state variable.

The controller specification includes the class that defines the agent controller, the name of each actuator, whether the actuator is operating in discrete or continuous space, the bounds on that actuator, and the number of discrete points in the case of discrete spaces (which include the minimum and maximum values in the range, with the remaining points evenly spaced within the range). For continuous actuators, the platform performs automatic rescaling post-processing from an assumed input range of $[-1, 1]$ to the desired actuator range, although this behavior can be disabled.

The default configurations are shown in Tables 3 and 4. In Table 4, $\mathcal{V}_{\text{safe}} = [0, 0.2 + 2n\sqrt{x^2 + y^2}]$ in 2D and $\mathcal{V}_{\text{safe}} = [0, 0.2 + 2n\sqrt{x^2 + y^2 + z^2}]$ in 3D.

Observation

The observation space can be represented using normalized magnitudes and transformed to platform reference frames, as summarized in the Rejoin Observation task example.

Magnitude-Normalized (MagNorm) Representation— Processing angles or orientations can be challenging for neural networks due to the discontinuity of angle values wrapping around the unit circle. An alternative representation, inspired by the OpenAI Gym Pendulum-v0 [3] environment, is to break a single angle, θ into two continuous values, $[\cos(\theta), \sin(\theta)]^T$. This logic has been extended to all vector quantities in the mag-norm transformation. Applying the cos, sin representation to this angle is equivalent to normalizing the original vector. Extending this logic to vectors, the MagNorm() function of vectors is defined as:

$$\text{MagNorm}(\mathbf{v}) = \left[\|\mathbf{v}\|, \frac{\mathbf{v}^T}{\|\mathbf{v}\|} \right] \quad (21)$$

Platform Reference Frame—As actions affect the environment relative to the state of the agent, it may be desirable to feed the agent observations from its own reference frame rather than the global reference frame. To transform a vector from the global reference frame to a platform’s reference frame, it may be necessary to either move the origin to the platform’s position/velocity, rotate the vector to the platform’s local reference frame, or both. In this manuscript, rotating a vector \mathbf{v} in the local reference frame of a simulation named entity, f_{oo} , is denoted $\mathbf{v}^{f_{oo}}$.

Observations—The observations fed to agents in the baseline solutions for the Aerospace SafeRL Framework benchmark tasks are shown in Tables 5 and 6. Each observation component is a vector or scalar quantity described in the rows of the observation tables. Each component is normalized and clipped with the elementwise normalization divisor shown in the Normalization Const column and the clipping bounds shown in the Clipping column. The normalized and clipped observation components are then concatenated into the final 1D observation vector and returned to the agent.

Reward

Tables 7 and 8 show individual reward component values for the Rejoin and Docking tasks.

Rejoin—The rejoin reward contains both sparse rewards that reward/punish success/failure at the end of the episode and dense rewards that provide immediate feedback during the episode. While the failure rewards are all constant, the success reward includes an episode length component to incentivize quicker solutions. The distance change reward is proportional to the change in an exponential potential function of the distance between the wingman and lead. The “In Rejoin” reward provides a constant reward every timestep the wingman is within the rejoin region. However, to prevent infinite reward cycles, where the agent can enter and exit the reward region multiple times while accumulating more reward, the cumulative “In Rejoin” reward is refunded if the wingman leaves the rejoin region before successfully completing a 20 second rejoin hold. The “In Rejoin First Time” gives a single reward in the timestep that the wingman enters the rejoin region for the first time during an episode. This reward component is helpful to incentivize the agent to enter the reward region early on in the training process.

Table 3. Rejoin Default Environment Configurations

	2D	3D
Lead		
Velocity (ft/s)	[200,400]	[200,400]
Velocity @ t_0 (ft/s)	[250,300]	[250,300]
x,y Position @ t_0 (ft)	[5000,10000]	[5000,10000]
z altitude @ t_0 (ft)	-	[12000,14000]
Heading @ t_0 (rad)	[0,2 π]	[0,2 π]
Wingman		
Velocity	[200,400]	[200,400]
State Reference	Lead Aircraft	Lead Aircraft
Rel. Velocity @ t_0 (ft/s)	[200,400]	[200,400]
Rel. Distance @ t_0 (ft)	[5000,10000]	[5000,10000]
Rel. Angle @ t_0 (rad)	[0,2 π]	[0,2 π]
z altitude @ t_0 (ft)	-	[10000,16000]
Heading Rate, $\dot{\psi}$ (rad/s)	[-0.17,0.17]	-
Roll Rate, $\dot{\phi}$ (rad/s)	-	[-0.17,0.17]
Flight Path Angle Rate, $\dot{\gamma}$ (rad/s)	-	[-0.09,0.09]
Acceleration, \dot{v} (ft/s ²)	[-96.5,96.5]	[-96.5,96.5]
Rejoin Region		
State Reference	Lead Aircraft	Lead Aircraft
Shape	Circle	Cylinder
Aspect Angle θ_{AA} (deg)	60	60
Relative Distance ρ_r (ft)	500	500
Region Radius ρ_e (ft)	150	100
Region Height h_e (ft)	-	300

Table 4. Docking Default Environment Configurations

	2D	3D	2D Oriented
Chief			
$x_0, \dot{x}_0, y_0, \dot{y}_0, z_0, \dot{z}_0$	0	0	0
Deputy			
Mass, m (kg)	12	12	12
Mean Motion, n (rad/s)	0.001027	0.001027	0.001027
State Reference	Chief	Chief	Chief
Velocity @ t_0 (m/s)	[0, $\mathcal{V}_{\text{safe}}$]	[0, $\mathcal{V}_{\text{safe}}$]	[0, $\mathcal{V}_{\text{safe}}$]
Rel. Distance @ t_0 (m)	[100,150]	[100,150]	[100,150]
Rel. Azimuth @ t_0 (rad)	[0,2 π]	[0,2 π]	[0,2 π]
Rel. Polar @ t_0 (rad)	-	[0, π]	-
Attitude, θ @ t_0 (rad)	-	-	[0, 2 π]
Angular Velocity, $\dot{\theta}$ (rad/s)	-	-	$[-\frac{\pi}{90}, \frac{\pi}{90}]$
Thrust X, F_x (N)	[-1,1]	[-1,1]	[-1,2]
Thrust Y, F_y (N)	[-1,1]	[-1,1]	-
Thrust Z, F_z (N)	-	[-1,1]	-
React. Wheel, $\ddot{\theta}$ (rad/s ²)	-	-	$[-\frac{\pi}{180}, -\frac{\pi}{180}]$
Docking Region			
State Reference	Chief	Chief	Chief
Shape	Circle	Sphere	Circle
x offset (m)	0	0	0
y offset (m)	0	0	0
z offset (m)	-	0	0
radius (m)	0.5	0.5	0.5

Table 5. Rejoin Observations

Description	Expression	Normalization Const	Clipping
2D Rejoin			
Lead Position in Wingman reference	$\text{MagNorm}((\mathbf{r}_l - \mathbf{r}_w)^w)$	[1000, 1, 1]	[-1, 1]
Rejoin Region Position in Wingman reference	$\text{MagNorm}((\mathbf{r}_{rejoin} - \mathbf{r}_w)^w)$	[1000, 1, 1]	[-1, 1]
Wingman velocity in Wingman reference	$\text{MagNorm}(\mathbf{v}_w^w)$	[400, 1, 1]	[-1, 1]
Lead velocity in Wingman reference	$\text{MagNorm}(\mathbf{v}_l^w)$	[400, 1, 1]	[-1, 1]
3D Rejoin			
Lead Position in Wingman reference	$\text{MagNorm}((\mathbf{r}_l - \mathbf{r}_w)^w)$	[1000, 1, 1]	[-1, 1]
Rejoin Region Position in Wingman reference	$\text{MagNorm}((\mathbf{r}_{rejoin} - \mathbf{r}_w)^w)$	[1000, 1, 1]	[-1, 1]
Wingman velocity in Wingman reference	$\text{MagNorm}(\mathbf{v}_w^w)$	[400, 1, 1]	[-1, 1]
Lead velocity in Wingman reference	$\text{MagNorm}(\mathbf{v}_l^w)$	[400, 1, 1]	[-1, 1]
Wingman Roll	ϕ_w	$[\frac{\pi}{3}]$	[-1, 1]
Wingman Flight Path Angle	γ_w	$[\frac{\pi}{9}]$	[-1, 1]

Table 6. Docking Environment Observations

Description	Expression	Normalization Const	Clipping
2D Docking			
Deputy Position	$\mathbf{r}_d = [x_d, y_d]$	[100, 100]	$[-\infty, \infty]$
Deputy Velocity	$\mathbf{v}_d = [\dot{x}_d, \dot{y}_d]$	[0.5, 0.5]	$[-\infty, \infty]$
Deputy Speed	$\ \mathbf{v}_d\ $	1	$[-\infty, \infty]$
Velocity Limit	$0.2 + 2n\ \mathbf{r}_d\ $	1	$[-\infty, \infty]$
3D Docking			
Deputy Position	$\mathbf{r}_d = [x_d, y_d, z_d]$	[100, 100, 100]	$[-\infty, \infty]$
Deputy Velocity	$\mathbf{v}_d = [\dot{x}_d, \dot{y}_d, \dot{z}_d]$	[0.5, 0.5, 0.5]	$[-\infty, \infty]$
Deputy Speed	$\ \mathbf{v}_d\ $	1	$[-\infty, \infty]$
Velocity Limit	$0.2 + 2n\ \mathbf{r}_d\ $	1	$[-\infty, \infty]$
2D Oriented Docking			
Lead Position in Deputy reference	$\mathbf{r}_l^d = [x_l, y_l]^d$	[100, 100]	$[-\infty, \infty]$
Deputy Velocity in Deputy Reference	$\mathbf{v}_d^d = [\dot{x}_d, \dot{y}_d]^d$	[0.5, 0.5]	$[-\infty, \infty]$
Deputy Orientation	$[\cos(\theta_d), \sin(\theta_d)]$	[-1]	$[-\infty, \infty]$
Deputy Angular Velocity	$\dot{\theta}_d$	$\frac{\pi}{90}$	$[-\infty, \infty]$
Deputy Speed	$\ \mathbf{v}_d\ $	1	$[-\infty, \infty]$
Velocity Limit	$0.2 + 2n\ \mathbf{r}_d\ $	1	$[-\infty, \infty]$

and counteract the “In Rejoin” refund within the discounted future reward estimate.

Docking—The docking rewards are very similar to the rejoin rewards described above with sparse rewards for success/failure terminal states and dense rewards for instantaneous feedback. The success and distance reward are identical to the rejoin rewards described above with slight changes to normalization parameters. Docking also has an additional velocity constraint reward with a constant penalty condition applied whenever the constraint is violated in addition to a scaling penalty that grows proportionally with the degree of constraint violation, i.e. violating the safety constraint is bad, violating it by a lot is very bad.

Terminal State

Tables 9 and 10 show the terminal states for the rejoin and docking problem respectively. The top row in both tables show the agent’s success condition and the subsequent rows show the various failure condition. Of particular note, the docking problem has a velocity constraint reward bound failure condition that terminates the episode when its reward reaches a predetermined lower bound of -5. This terminal condition creates a hard limit on the soft velocity constraint, preventing reward from growing negatively unbounded and pruning episodes that are clearly unable to respect this soft constraint at all. The reward is used as a proxy to measure degree of constraint compliance failure, although this terminal state does not necessarily need to be directly coupled to reward and is done so here for simplicity. The soft velocity constraint can be tightened and even made hard by decreasing the reward lower to zero.

Implementation details

For the results shown in Section 6, both environments used a discrete action space, normalized observation space, and a batch size of 4000. To train the rejoin, truncated episodes were used in each batch, while the docking training only used complete episodes in each batch.

Logging

The RLlib framework supports callbacks, which implement methods that are called periodically throughout training (once every environment step, once every episode, and once during postprocessing of trajectories). A CallbacksCaller class was created to maintain a list of callback instances, allowing for the use of multiple callbacks in a single training run. Some callbacks create custom metrics desired for training analysis, such as episode outcome (success or failure), failure code, reward components, or statuses. The LoggingCallback is responsible for the creation of logs. On construction, the logging callback consumes information from the training script regarding the number of rollout workers producing logs, the logging interval (the number of episodes to omit between logged episodes), and logging verbosity. During training, the LoggingCallback’s `on_episode_step()` method records simulation state data from the episode and to the desired log file. One log file is created per logging worker. The data is stored in the JSONlines format, to maintain readability and portability to other platforms.

Plotting

To gain deeper insights into policy performance, there are two primary methods for plotting training episodes: Tensorboard and Jupyter Notebook. Tensorboard may be used to plot common reinforcement learning training metrics such as total

environment steps, rewards, total loss, and more. Custom metrics such as episode outcomes, failure code rates, and total reward components may also be plotted on Tensorboard. All plots have an adjustable x axis, allowing the user to choose between environment steps, relative, or wall clock time as the x variable.

For more specific insights into training, users may alter the `plotting_notebook` Jupyter Notebook included in the framework at `scripts/visualization/plot_notebook.ipynb`. This notebook, holds a small tutorial for analyzing training results and allows users access to `matplotlib` for advanced visualization. This tool is useful for users with experience in python who want to visualize specific custom details of their policy’s interactions with its environment during training.

Scripts

Training script—The included training script, `scripts/train.py`, can be used to construct an environment and train an agent directly from a yaml config file. This training script uses Ray RLlib and Ray Tune, however the environments can be integrated into any OpenAI Gym compatible RL framework. This training script offers many options for customizing training. These options include specifying the environment configuration, altering the length of training, determining whether the policy is trained on batches of truncated or complete episodes, changing the location of the output directory, specifying the number of rollout workers and how many of them produce logs of their episodes, altering the frequency of checkpoints, configuring mid training evaluation episodes, and more.

Evaluation script—The `scripts/eval.py` script, may be used to run post hoc evaluation rollouts to benchmark the performance of your trained policy. This script will load in the policy and environment with the same configuration from training and run ten rollout episodes. There are a number of command line options that can be used to tailor the eval script’s function. With these command line options, one can select a specific saved checkpoint of the policy to evaluate, pass a specific seed to the environment, determine the output directory, alter the number of evaluation episodes to perform, alter the configuration of the environment to test policy generalizability, and turn on rendering to gain visual intuition of policy behavior and performance.

Tests

A suite of tests was developed leveraging `pycharm` to verify the most important features of the codebase. All tests are written in `pycharm` and organized into three major categories: unit tests, integration tests, and system tests. Targeted unit tests are used to ensure the function of vital modules. Integration tests are used to ensure interactions between modules behave as expected. System tests are used to ensure the high level function of the core features (namely the training of agents on key environments). These tests give developers quick methods for verifying unintended behavior is not introduced during the creation of new features. Tests help to catch code defects earlier in the development cycle, ultimately reducing development time.

ACKNOWLEDGMENTS

This material is based upon work supported by the Air Force Research Laboratory Innovation Pipeline Fund. In particular the authors would like to thank Sean Mahoney whose support

Table 7. Rejoin Reward

Description	Expression
Distance Change	$R_t^d = 2(e^{-ad_t} - e^{-ad_{t-1}})$ $d_i = \ \mathbf{r}_{w,i} - \mathbf{r}_{l,i}\ $ $a = \frac{\ln(2)}{5000}$
In Rejoin (refunded on exiting Rejoin Region)	$R_t^r = \begin{cases} +0.1 & , \text{if } d_t^r \leq 150 \\ -\sum_{i=0}^{t-1} R_i^r & , \text{if } d_t^r > 150 \end{cases}$ $d_t^r = \ \mathbf{r}_{w,t} - \mathbf{r}_{rejoin,t}\ $
In Rejoin First Time (once per episode)	$R_t^f = \begin{cases} +0.25 & , \text{if } (d_t \leq 150) \wedge (\forall i \in [0, t-1] : d_i > 150) \\ 0 & , \text{if } d_t > 150 \\ 0 & , \text{if } \exists i \in [0, t-1] : d_i \leq 150 \end{cases}$ $d_t = \ \mathbf{r}_{w,t} - \mathbf{r}_{rejoin,t}\ $
Success	$+2 - \frac{t}{t_{max}}$
Failure	
Crash	-1
Distance	-1
Timeout	-1

Table 8. Docking Reward

Description	Expression
Distance Change	$R_t^d = 2(e^{-ad_t} - e^{-ad_{t-1}})$ $d_i = \ \mathbf{r}_{deputy,i} - \mathbf{r}_{chief,i}^i\ $ $a = \frac{\ln(2)}{100}$
Velocity Constraint	$R_t^{vc} = \begin{cases} -0.01 - 0.01(\ \mathbf{v}_{deputy}\ - v_{limit}) & , \text{if } \ \mathbf{v}_{deputy}\ > v_{limit} \\ 0 & , \text{if } \ \mathbf{v}_{deputy}\ \leq v_{limit} \end{cases}$
Δv	$-0.01(\ \frac{\mathbf{u}}{m}\)$
Success	$+2 - \frac{t}{t_{max}}$
Failure	
Crash	-1
Distance	-1
Timeout	-1
Vel Constr. Reward Limit	0

Table 9. Rejoin Terminal States

Description	Condition
Success	$\forall i \in [t-20, t], \ \mathbf{r}_{w,i} - \mathbf{r}_{rejoin,i}\ \leq 150$
Crash	$\ \mathbf{r}_{w,t} - \mathbf{r}_{l,t}\ \leq 100$
Distance	
2D	$\ \mathbf{r}_{w,t} - \mathbf{r}_{l,t}\ > 40000$
3D	$\ \mathbf{r}_{w,t} - \mathbf{r}_{l,t}\ > 100000$
Timeout	
2D	$t > 1000$
3D	$t > 2000$

Table 10. Docking Terminal States

Description	Condition
Success	$\ \mathbf{r}_d - \mathbf{r}_c\ \leq 0.5, \ \mathbf{v}_d\ \leq v_{limit}$
Crash	$\ \mathbf{r}_d - \mathbf{r}_c\ \leq 0.5, \ \mathbf{v}_d\ > v_{limit}$
Distance	$\ \mathbf{r}_d - \mathbf{r}_c\ > 40000$
Timeout	
2D/3D	$t > 2000s$
2D Oriented	$t > 3000s$
Vel Constr. Reward Limit	$\sum_{i=0}^t R_i^{vc} < -5$

contributed to the success of this project. The views expressed are those of the authors and do not reflect the official guidance or position of the United States Government, the Department of Defense or of the United States Air Force.

REFERENCES

- [1] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, Jan. 2016.
- [2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, “Grandmaster Level in StarCraft II using Multi-Agent Reinforcement Learning,” *Nature*, vol. 575, no. 7782, pp. 350–354, Oct. 2019.
- [3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.
- [4] J. Garcia and F. Fernández, “A comprehensive survey on safe reinforcement learning,” *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.
- [5] A. Ray, J. Achiam, and D. Amodei, “Benchmarking safe exploration in deep reinforcement learning,” *arXiv preprint arXiv:1910.01708*, vol. 7, 2019.
- [6] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, “Rllib: Abstractions for distributed reinforcement learning,” in *International Conference on Machine Learning*. PMLR, 2018, pp. 3053–3062.
- [7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.
- [8] L. Graesser and W. L. Keng, *Foundations of deep reinforcement learning: theory and practice in Python*. Addison-Wesley Professional, 2019.
- [9] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, “Safe reinforcement learning via shielding,” in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [10] J. G. Rivera, A. A. Danylyszyn, C. B. Weinstock, L. R. Sha, and M. J. Gagliardi, “An architectural description of the simplex architecture,” Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, PA, Tech. Rep., 1996.
- [11] T. Gurriet, M. Mote, A. Singletary, P. Nilsson, E. Feron, and A. D. Ames, “A scalable safety critical control framework for nonlinear systems,” *IEEE Access*, vol. 8, pp. 187 249–187 275, 2020.
- [12] T. Gurriet, M. Mote, A. D. Ames, and E. Feron, “An online approach to active set invariance,” in *2018 IEEE Conference on Decision and Control (CDC)*. IEEE, 2018, pp. 3592–3599.
- [13] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada, “Control barrier functions: Theory and applications,” in *2019 18th European Control Conference (ECC)*. IEEE, 2019, pp. 3420–3431.
- [14] M. Nagumo, “Über die lage der integralkurven gewöhnlicher differentialgleichungen,” *Proceedings of the Physico-Mathematical Society of Japan. 3rd Series*, vol. 24, pp. 551–559, 1942.
- [15] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep reinforcement learning that matters,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.
- [16] L. E. Dubins, “On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents,” *American Journal of mathematics*, vol. 79, no. 3, pp. 497–516, 1957.
- [17] W. Clohessy and R. Wiltshire, “Terminal guidance system for satellite rendezvous,” *Journal of the Aerospace Sciences*, vol. 27, no. 9, pp. 653–658, 1960.
- [18] G. W. Hill, “Researches in the lunar theory,” *American journal of Mathematics*, vol. 1, no. 1, pp. 5–26, 1878.
- [19] T. McLain, R. W. Beard, and M. Owen, “Implementing dubins airplane paths on fixed-wing uavs,” 2014.
- [20] M. L. Mote, C. W. Hays, A. Collins, E. Feron, and K. L. Hobbs, “Natural motion-based trajectories for automatic spacecraft collision avoidance during proximity operations,” in *2021 IEEE Aerospace Conference*. Institute of Electrical and Electronics Engineers (IEEE), 2021, pp. 1–12.
- [21] K. Dunlap, M. Mote, K. Delsing, and K. L. Hobbs, “Run Time Assured Reinforcement Learning for Safe Satellite Docking,” *AIAA SciTech Forum*, , Submitted.

- [22] C. D. Petersen, S. Phillips, K. L. Hobbs, and K. Lang, “Challenge problem: Assured satellite proximity operations,” in *31st American Astronautical Society Space Flight Mechanics Meeting*. American Astronautical Society (AAS), 2021, pp. 1–20.

BIOGRAPHY



Umberto Ravaioli is an Analyst at Toyon Research Corporation and the AI Lead on the Safe Autonomy Team at Air Force Research Laboratory’s Autonomy Capability Team (ACT3). There, he leads the overall effort of designing/developing the Aerospace SafeRL Framework, training RL agents, and building Runtime Assurance algorithms. At Toyon, he has contributed to DOD research efforts including Deep Learning Aerial Target Recognition with EO+LIDAR fusion, image based helicopter engine predictive maintenance, and GPS denied vision-aided navigation. Umberto received his BS in Electrical Engineering and MS in Computer and Electrical Engineering from the University of Illinois at Urbana-Champaign.



James Cunningham is a Junior Autonomy Engineer on the Autonomy Capability Team (ACT3) at the Air Force Research Laboratory. There he works on the Safe Autonomy team studying approaches towards safety in autonomous systems which use learning-based control methods. His previous experience includes work in efficient parameter-free online clustering, multi-domain learning-based data embedding and retrieval, automatic supervised large-scale dataset generation, autonomous tracking in Wide Area Motion Imagery, and discriminatory algorithms in the SAR domain. James received his BS and MS degree in Computer Science and Engineering from Ohio State University.



John McCarroll is a Junior Autonomy Engineer on the Autonomy Capability Team (ACT3) at the Air Force Research Laboratory. He works on the Safe Autonomy team, supporting development efforts and experimenting with reinforcement learning. John received his BS degree in Software Engineering from Rochester Institute of Technology, where he gained experience in neural architecture search using genetic algorithms, CNNs applied to games and art, data warehousing, and data visualization.



Vardaan Gangal is a Junior Autonomy Engineer on the Autonomy Capability Team (ACT3) at the Air Force Research Laboratory. There he works on the Safe Autonomy team studying approaches towards safety in autonomous systems which use learning-based control methods. Vardaan received his BS degree in Computer Science and Engineering from Ohio State University.



Kyle Dunlap is a graduate student in Aerospace Engineering at the University of Cincinnati. His research interests are Real-Time Safety Assurance, Safety Critical Control, Autonomous Systems, Reinforcement Learning, and Genetic Fuzzy Systems. His previous experience includes developing control-based fault detection and mitigation strategies for electrified aircraft propulsion systems, applying deep reinforcement learning to safe autonomous spacecraft docking, and developing real-time safety assurance techniques for spacecraft docking. Kyle received his BS in Aerospace Engineering from the University of Cincinnati.



Kerianne Hobbs is the Safe Autonomy Lead on the Autonomy Capability Team (ACT3) at the Air Force Research Laboratory. There she investigates rigorous specification, analysis, and bounding techniques to enable certification of autonomous and learning controllers for aircraft and spacecraft applications. Her previous experience includes work in automatic collision avoidance and autonomy verification and validation research. Kerianne has a BS in Aerospace Engineering from Embry-Riddle Aeronautical University, an MS in Astronautical Engineering from the Air Force Institute of Technology, and a Ph.D. in Aerospace Engineering from the Georgia Institute of Technology.