CEA Comité Español de Automática
XLVII JORNADAS DE AUTOMÁTICA CÓRDOBA 2026
UNIVERSIDADE DA CORUÑA

Jornadas de Automática

A platform for the navigation of Unmanned Aerial Vehicles based on Reinforcement Learning

Álvarez Novoa, Daniel[a], Pérez-Muñoz, Ángel-Grover[a,*], Alonso, Alejandro[a], Pérez, María S.[b]

[a] *Information Processing and Telecommunications Center, E.T.S. de Ingenieros de Telecomunicación, Universidad Politécnica de Madrid, Spain*
[b] *Ontology Engineering Group, ETSI de Informáticos, Universidad Politécnica de Madrid, Spain*

To cite this article: Álvarez Novoa, D., Pérez-Muñoz, Á.G., Alonso, A., Pérez, M.S. 2026. A platform for the navigation of Unmanned Aerial Vehicles based on Reinforcement Learning Jornadas de Automática, 47. https://doi.org/10.17979/ja-cea.2026.47.13789

Resumen

Este artículo presenta una plataforma para el desarrollo de casos de uso para el control automatizado de vehículos aéreos no tripulados (*UAV*), utilizando el simulador AirSim. Esta plataforma permite generar escenarios de vuelo realistas con múltiples UAV. La plataforma propuesta facilita la creación de casos de uso para el desarrollo, validación y verificación del *Reinforcement Learning* (*RL*) y las redes neuronales en sistemas críticos en tiempo real. Estos algoritmos pueden aprender a navegar en entornos dinámicos sin intervención humana. La plataforma se ha validado utilizando dos UAV con redes neuronales entrenadas mediante el algoritmo *Soft-Actor-Critical* (*SAC*).

Palabras clave: Control mediante aprendizaje por refuerzo, Vehículos autónomos, Plataforma para aprendizaje y validación

Una plataforma para la navegación de vehículos aéreos no tripulados basada en el aprendizaje por refuerzo

Abstract

This article introduces a platform for developing use cases for the automated control of unmanned aerial vehicles (*UAVs*), utilising the AirSim simulator. This platform allows for the generation of realistic flight scenarios involving multiple UAVs. The proposed platform facilitates the construction of use cases for the development, validation, and verification of reinforcement learning (*RL*) and neural networks in critical real-time systems. These algorithms can learn to navigate dynamic environments without human intervention. The platform was validated using two UAVs with neural networks that were trained using the Soft Actor-Critical (*SAC*) algorithm.

Keywords: Reinforcement learning control, Autonomous Vehicles, Platform for learning and validation

1. Introduction

There is significant interest in applying machine learning (ML) in safety-critical systems. Compared to equivalent current code, ML is expected to offer more efficient computation times and greater adaptability to unexpected events. Safety-critical systems are defined as systems in which life, physical integrity, or the environment are at risk in the event of a failure (Rierson, 2013). Deploying a safety-critical system requires qualification or certification by regulatory authorities. However, currently available ML techniques do not meet the stringent safety standards required.

This paper focuses on the application of ML techniques in safety-critical systems, including the identification of safety requirements that may preclude their use, the selection of appropriate ML approaches, and the development of neural networks for embedded systems. To this end, a number of use cases have been examine to propose activities involving ML techniques in safety systems:

*Autor para correspondencia: angel.perez.munoz@upm.es

- the attitude control of a satellite, based on a simulator used in the the UPMSat-2 satellite (Pérez-Muñoz et al., 2025b).
- the active thermal control of a sensitive radiometer, to be aboard the UPMSat-3 micro-satellite.
- the control of a drone navigation and collision avoidance system, which relies on the AIRsim simulator (Pérez-Muñoz et al., 2025a).

Artificial Neural Networks (ANNs) are a type of ML model whose structure is inspired by biological neurons found in animal brains. The most basic configuration for ANNs is the multilayer perceptron (MLP), which consists of different layers, with each neuron in one layer connected to the next. Overall, ANNs consist of matrix multiplications that can be programmed as non-recursive functions, thus avoiding the need for branching statements such as loops and conditional branches. Compared to other online, optimisation-based ML techniques, ANNs offer predictable timing, which is essential for safety-critical systems.

This work presents a platform for investigating the application of ML techniques in safety systems. The simulation environment uses AirSim, which incorporates external elements alongside drones. Developed by Microsoft, AirSim is an open-source simulator designed to support the development of autonomous vehicles, including drones. It supports fine-grained control and a high sampling frequency, with a minimum simulation period of 1 ms. Various sensors that can be used with drones to provide information such as their position, velocity and attitude are simulated. Drone movements result from thrust and torques generated by motor voltages, as well as external forces such as drag, friction, and gravity. The tool provides a Python API that allows external software to control drones and access sensor data.

This work aims to design and implement a platform that facilitates the creation, operation and training of multiple drones, as well as validating their behaviour. The initial implementation is inflexible, making it difficult to create and configure similar drones with different controllers. This platform has been used to develop an ANN to control the navigation of a drone. We are currently interested in creating use cases involving multiple drones.

2. State of Art

Reinforcement Learning (RL) is a method where an agent learns an optimal behaviour by interacting with a dynamic environment. The learning aims to improve the agent so that it can choose actions that maximize rewards accumulated over a period of time (Sutton and Barto, 2018). As shown in Figure 1, its main elements are the agent, the action, the state, the environment, and the reward.

The agent contains the policy that acts as the controller of the system, and the training or learning algorithm to update the policy. The policy is typically implemented using neural networks. The actions are the commands made by the agent given a state, which may be discrete (e.g. turn on/off actuator) or continuous (e.g. applied energy on the actuator). The state represents the status of the environment. In the case of partial observability, the agent is limited to the observable states.

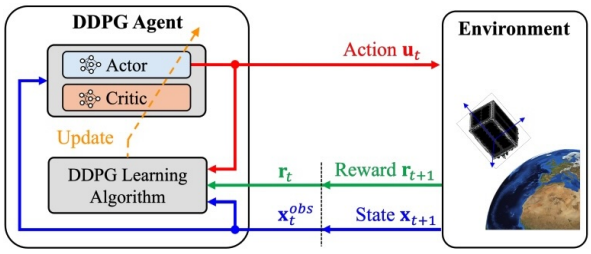

Figure 1: Reinforcement learning elements.

The environment contains all external elements, including the plant, dynamics, sensors, and actuators. It receives the action from the agent and responds with a new state and reward. The reward is a numerical value that criticizes the agent's performance based on the state of the environment, with higher values indicating better performance. The reward and state are used by the training algorithm to update the policy so that the actual and future rewards are maximized.

For this use case, we selected the Soft Actor-Critic (SAC) learning algorithm (Haarnoja et al., 2018). SAC implements the policy as a neural network (the actor) receiving the state and returning the actions. It includes more neural networks (the critics) that evaluate the actor's improvement during training by estimating the cumulative reward, which indicates how good a given action is in a particular state.

The deployment of a safety-critical system requires its qualification or certification by regulatory authorities or by law (Rierson, 2013). This characteristic is assessed by requiring the system (and software) to be validated to strict standards covering all aspects of requirements definition, design, production, verification and validation, operation, and maintenance.

The ECSS is an initiative established to develop a coherent, single set of user-friendly standards for use in all European Space activities. It is a cooperative of the European Space Agency, national space agencies, and European industry associations. Specifically, the analysis was performed on the following standards:

- Space engineering. Software (ECSS-E-ST-40C, Rev.1 DIR1). This document is not yet approved. It is in the "public review" phase. This is the main basis on this analysis (ECSS, 2021).
- Space product assurance. Software product assurance. (ECSS-Q-ST-80C Rev.1) (ECSS, 2017b).
- Space product assurance. Safety (ECSS-Q-ST-40C Rev.1) (ECSS, 2017a).

3. Software architecture of the platform

The high-level software architecture in Figure 2 shows the most important components and their relationships. This architecture aims to be flexible for the creation and simulation of different types of drones. It also aims to support both the training and validation processes. The main components in the architecture include:

- Scenery generator: Generates a set of scenarios used to configure the environment in the training and validation processes. The configuration of the environment allows these processes to manage the generation of the data.
- RL training & validation: Manages the training and testing procedures of the RL agent (neural networks).
- Environment: it provides an API to create, initialize, and start the simulation of drones in the system.

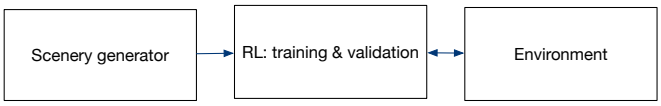

Figure 2: Reinforcement learning elements.

3.1. Scenery generator component

Supervised learning requires datasets as the main input to train and produce the ANN. When dealing with RL, data are generated online through the interactions between the RL agent and the environment. Then, rather than predefining static datasets, a set of scenarios is defined to configure the environment and drive the data generation process.

Each scenario contains information to configure the environmental parameters. Scenarios support two types of configurations. First, configuration for parameters assigned with fixed values that are maintained throughout the complete execution. Finally, configurations that describe the behaviour of dynamic elements, such as the strategy followed by a defender drone to avoid collisions.

As depicted in Figure 3, the scenery generator component is a software process that produces the scenarios that configure the environment. This process is executed before training or validating the neural network because they require the presence of scenarios to start.

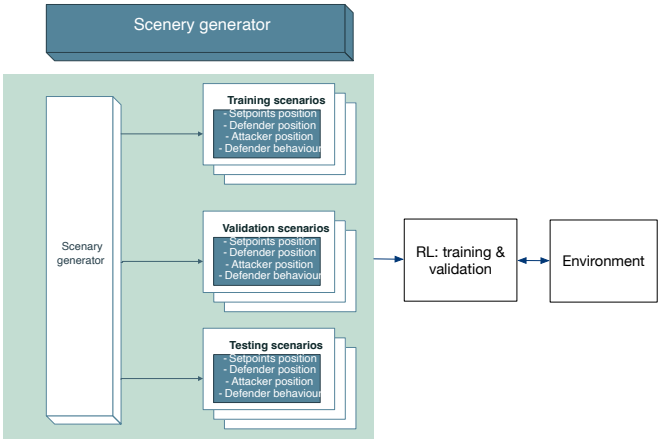

Figure 3: Elements inside the scenery generator component.

The generated scenarios are divided into three sets following the same criteria as in SL:

- Training scenarios: used during the learning process to update the neural network parameters.
- Validation scenarios: used in training to evaluate the network performance and determine the most suitable configuration.
- Testing scenarios: used to assess the final behaviour and generalization of the trained network within the corresponding validation software environment.

Adequate scenarios must satisfy three main requirements:

- Completeness: ensures that the scenarios generated cover all relevant operational situations
- Representativeness: guarantees that the scenarios resemble realistic operational conditions.
- Independence: that information from one dataset does not leak into another.

In order to achieve these requirements, the scenario generator can be parameterized using the number of training, validation, and testing scenarios, together with a random seed to ensure reproducibility. Completeness is achieved by defining all relevant variables that may influence the drone's behavior. In this case, these variables include the defender position, attacker position, setpoint positions, and defender strategy. Representativeness and independence are ensured by randomly sampling non-repeated parameter combinations according to a uniform distribution.

Finally, all generated scenarios are divided into three independent files corresponding to the training, validation, and testing datasets. These files are loaded when required during the training or evaluation stages.

3.2. Training component

The training process relies on the SAC algorithm. Its implementation relies on the Stable Baselines3 (DLR, 2026) library, that provides reliable open-source implementations of deep reinforcement learning (RL) algorithms in Python. It allows to define the neural network architecture and the associated hyperparameters. The provided algorithms follow a consistent interface and extensive documentation, for facilitating the train and compare different RL algorithms.

The training development relies on a set of scenarios created, which are processed sequentially. Each scenario is executed of loops where each step is processed, the reward output is analysed and the ANN is updated. When the scenario is finished, it is compared the previous and current ANNs, based on the validation scenarios. The best is selected for its improvement in the next scenario. When all the scenarios have been processed, the best ANN is considered to the output in the training phase.

During training, periodic evaluation episodes are also executed. In these episodes, the neural network operates without updating its parameters, and its performance is compared against the best previous results. If better performance is achieved, the new model is stored as the current best neural network.

Once all training steps are completed, the trained agent is saved for later deployment in the evaluation software. The collected rewards and logs are also stored for further analysis and training performance visualization.

Stable Baselines3 uses the Gymnasium, which is an API standard for reinforcement learning with a diverse collection of reference environments. In particular, the enviroment component provides this interface, which is extensive used the training component.

The learning process relies on the functions provided by the environment component, according to this API:

- Use of `init(scenarios)`: generated scenarios are loaded, and the connection to AirSim is initialized together with the corresponding drones and setpoints.
- Use of `reset()`: loads a new scenario at the beginning of each episode, updating the positions of both drones and setpoints, and selecting the defender's behavior. It then returns the initial observation to the agent.
- Use of `step(action)`: the training loop starts by resetting the environment and obtaining the initial observation. Based on this observation, the agent computes the action to be applied to the environment function. This function returns the new observation and the associated reward. Using this information, the reinforcement learning algorithm updates the neural network parameters when required and computes the next action. This process is repeated iteratively until the episode is completed.
- Use of `close()`: terminates the communication with the environment component.

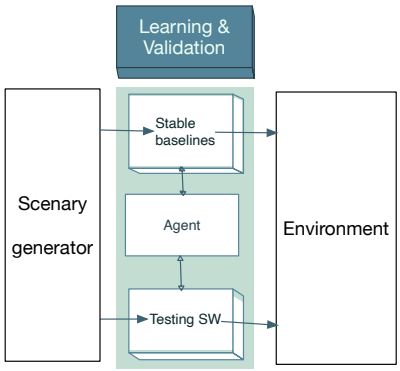

Figure 4: Elements inside the Learning & Validation component.

3.3. Environment component

This component relies on the following sub-components:

- Environment manager: this drives the behaviour of the environment. It gets commands from the training component and interacts with the AirSIM for its execution.
- Drones objects: these objects provides the required information for the training and testing phases. There is a relation one-to-one with the drones in the simulator, which updates its configuration on each step. It also evaluates the reward of a step, as an output for the training components.
- AirSIM simulator:(Shah et al., 2018). AirSim provides a Python interface to control the drone's velocity and to sense the drone's position, velocity, and acceleration. The environment manager invokes the API of this tool, for its simulation.

This environment implements the Gymnasium API. In the previous section we described how the training software used this API. Here, we change the perspective and describe the implementation of each function inside the environment:

- `init(scenarios)`: configures and creates the required objects in the simulator. The global information on the scenarios is the input for this function.
- The `reset()` function loads a new scenario at the beginning of each episode, updating the positions of both drones and setpoints, and selecting the defender's behavior. It then returns the initial observation to the agent.
- The `step(action)` function is executed at every control cycle. It ensures synchronization of all drones within the environment by pausing AIRSIM execution and commanding the non-learning drones (in this case, the defender) to perform their predefined actions. Simultaneously, it applies the action computed by the agent to the attacker drone. Finally, it returns the updated observations and the corresponding reward to the training or validation software at the end of the time step.
- The `close()` function terminates the communication with AIRSIM.

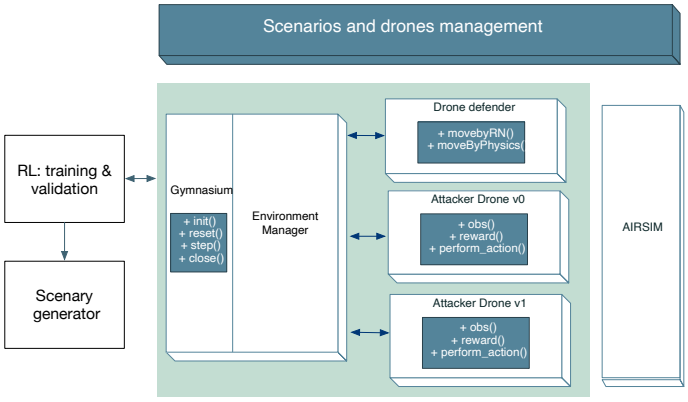

Figure 5: Elements inside the Environment component.

4. Illustrative example of the platform

In this section, we use an illustrative example to demonstrate the application of the platform presented in Section 3. The presentation of this example progresses across the main stages of the development: system description, scenario definition, agent-environment interface, training, and validation. The discussion will also map the involvement of the software architecture in the development process.

4.1. System description

The illustrative example consisted of an attacker drone controlled by a Reinforcement Learning (RL) agent. The objective of the RL agent was to learn a control policy that allowed the attacker to collide with a defender drone. To achieve this objective, the RL agent had to ensure that the attacker maintained continuous visibility of the defender throughout the pursuit. Particularly, the RL agent controlling the attacker had to:

- Collide against a defender drone. The collision is satisfied when the distance between the attacker and defender is lower than or equal to 1.2 m.
- Maintain visibility of the defender. Visibility is ensured when the defender is within an ellipsoidal field of view centred at the attacker drone position with semi-axes lengths of $X = 48$ m, $Y = 42$ m, and $Z = 13$ m.

The Figure 6 illustrates the vision zone of the attacker drone. Two cases are exemplified. The first one with the defender outside the field of view of the attacker ($\mathbf{D}_{\text{not visible}}$). This is not desired because the attacker will have no visibility of the defender and will not be able to attempt collisions. The second situation is when the defender is within the attacker's vision zone ($\mathbf{D}_{\text{visible}}$).

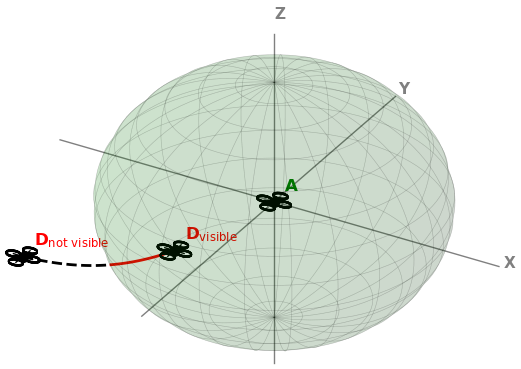

Figure 6: Vision zone of the attacker drone (**A**) showcasing the defender (**D**) inside and outside the attacker's field of view.

4.2. Scenarios definition

The first stage was to define the scenarios that configure the environment. Different environmental parameters were identified and configured according to their operational ranges. The main environmental parameters are as follows:

- *Initial position of the defender drone*. The initial position of the defender drone was generated following a uniform distribution. The X and Y coordinates were fixed at $(0, 0)$. The Z coordinate ranged from 6 m to 30 m above ground.
- *Initial position of the attacker drone*. The initial position of the attacker was restricted to ensure both visibility of the defender drone and no initial collision condition. Visibility was ensured by conditions illustrated in Figure 6. The initial distance to the defender was of at least 7.5 m.
- *Route followed by the defender drone*. The defender drone followed one out of four predefined routes composed of 18 setpoints. The trajectories covered positive and negative directions in the X and Y axes, while the altitude of each setpoint ranged from 6 m to 30 m above ground.
- *Navigation and anti-collision strategy of the defender*. The defender drone could follow two different strategies: a deterministic controller that followed the predefined setpoints without collision avoidance, or a neural network-based controller trained in previous work by the authors.
- *Maximum speed of the defender and the attacker*. The maximum velocity of both drones ranged from ±1 m/s to ±5 m/s in the X, Y, and Z axes.

The described conditions were coded inside the "Scenery Generator" component (Section 3.1, Figure 3). The samples were generated according to the predefined ranges. To ensure completeness and representativeness, these parameters were generated following a uniform distribution. Finally, to ensure independence between datasets, the scenarios were first generated without repetition. Also important was to preserve the uniformity of the parameter distributions across subsets.

By encapsulating these boundaries within the "Scenery Generator", operational constraints such as shifting from a uniform to a Gaussian distribution or changing speed limits can be modified through configuration files. This eliminates the need to alter the training loop or core simulation code, ensuring decoupled maintenance.

4.3. Interface with the environment

After the definition of scenarios, we defined the interface with the environment. The interface is comprised of three elements: the state observed by the RL agent, the action applied to the environment, and the reward signal used by the RL agent to evaluate its control policy.

The definition of these elements was subject to different iterations. This is because the RL agent performance depends strongly on the elements inside the state, action, and reward. This iterative workflow was supported by the proposed software architecture within the "Environment" component (Section 3.3, Figure 5). This architecture enabled the inclusion of different versions of the attacker drone, each defining its own state, action, and reward.

The best performing configuration was as follows. The state space was composed of the attacker velocity, the relative position of the defender with respect to the attacker, and the relative velocity between both drones:

$$s = [v_a, ; p_d - p_a, ; v_d - v_a] \in \mathfrak{R}^9, \tag{1}$$

where p_a and p_d represent the positions of the attacker and defender drones, respectively, while v_a and v_d denote their velocities. The action space consisted of the velocity commands applied to the attacker drone in the X, Y, and Z axes:

$$a = [v_x, ; v_y, ; v_z] \in \mathfrak{R}^3 \tag{2}$$

Finally, the reward function $r \in \mathfrak{R}$ encouraged the attacker to approach and collide with the defender. It also penalized losing visibility of the defender:

$$r = \begin{cases} +10 & \text{if } d < 1.2 \text{ m}, \\ -1 & \text{if out of vision zone}, \\ \dfrac{-2\,(p_d - p_a)^2}{60^2} - 0.1 & \text{otherwise.} \end{cases} \quad (3)$$

4.4. Training and evaluation

In this section we discuss the training and validation of the RL agent with a main focus on the usage of the proposed software architecture. The training configuration is based on a SAC RL agent that follows a multi-layer perceptron architecture, where both the actor (policy neural network) and critic (Q-function neural network) use two hidden layers with 64 neurons each and ReLU activation functions. Training was performed using standard SAC hyperparameters, including a learning rate of 0.001, discount factor of 0.99, batch size of 128, and an automatically tuned entropy coefficient to balance exploration and exploitation. A fixed random seed was used to ensure reproducibility.

After training, the evaluation process began. It consisted of an iterative loop that interacted with the environment through the standard Gymnasium API methods: `init(scenarios)`, `reset()`, `step(action)`, and `close()`. The testing workflow started by initializing the connection to AirSim and configuring the corresponding drones and set-points. The evaluation loop then started the environment and obtained the initial observation. Based on this observation, the agent computed the action to be executed in the environment and received the updated observation and the associated reward. This information was used to compute subsequent actions until the episode terminated. Once all the evaluation episodes were completed, all the rewards and logs were saved in order to plot the results and analyze the performance.

Training and evaluation were implemented in the Learning & Validation component described in Section 3.2. Although they correspond to distinct execution flows, both processes share a substantial amount of information like the dependency on the environment. These workflows were supported by the proposed software architecture, which promoted component reuse, consistency across pipelines, and reduced redundancy in data handling (see Figure 4).

5. Conclusions

To conclude, the proposed drone system provides a reproducible RL framework based on pre-generated scenarios satisfying completeness, representativeness, and independence criteria. The modular architecture enables integration of different drone types with minimal modifications, while the environment guarantees synchronized execution and time-consistent interaction.

In addition, the separation between the training and validation software and the rest of the system components supports deployment across different ML frameworks and validation setups. For instance, if increased realism is required through the use of embedded computing platforms, only minimal changes to the software would be necessary, as these modifications would be localized within the testing module and would not affect the rest of the system.

As a limitation, the proposed software architecture highly depends on the agent–environment interface. We consider that this interface should be further abstracted into a reusable component. In this way, any future modification to the interface would be isolated to a single component, improving maintainability and reducing the impact of changes across the software platform.

Overall, the system offered a modular and scalable architecture for the development and evaluation of drone control policies, facilitating extensibility, reproducibility, and deployment across different control software implementations.

Acknowledgements

This work was developend under the project Re-InITS *Reliable Industrial Information Technology Systems in the Computing Continuum Era* (PID2024-155230OB-C43) with financial support of the Ministerio de Ciencia e Innovación (Spain). We also thank the collaboration with project partners.

References

DLR, 2026. Stable baselines3, institute of robotics and mechatronics. URL: `https://www.dlr.de/en/rm/research/publications-and-downloads/software/stable-baselines3`.

ECSS, 2017a. European Cooperation for Space Standardization. ECSS-Q-ST-40C Space product assurance — Safety.

ECSS, 2017b. European Cooperation for Space Standardization. ECSS-Q-ST-80C Rev.1 – Space Product Assurance — Software Product Assurance.

ECSS, 2021. European Cooperation for Space Standardization. ECSS-E-ST-40C Rev.1, Dir.1, Space engineering — Software.

Haarnoja, T., Zhou, A., Abbeel, P., Levine, S., 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. CoRR abs/1801.01290. URL: `http://arxiv.org/abs/1801.01290`, doi:`10.48550/arXiv.1801.01290`, `arXiv:1801.01290`.

Pérez-Muñoz, A.G., López-García, G., García-Quijano, H., Alonso, A., 2025a. Development and validation of a safe reinforcement learning drone controller. Jornadas de Automática `10.17979/ja-cea.2025.46.12154`.

Pérez-Muñoz, A.G., López-García, G., García-Villoria, I., Alonso, A., Porras-Hermoso, A., Pérez, M.S., 2025b. Feasibility of deep reinforcement learning for the real-time attitude control of a satellite system. Journal of Systems Architecture 167, 103513. URL: `https://www.sciencedirect.com/science/article/pii/S1383762125001857`, doi:`https://doi.org/10.1016/j.sysarc.2025.103513`.

Rierson, L., 2013. Developing safety-critical software. CRC Press, Boca Raton, FL.

Sutton, R.S., Barto, A.G., 2018. Reinforcement Learning: An Introduction, 2nd edition. A Bradford Book, Cambridge, MA, USA.