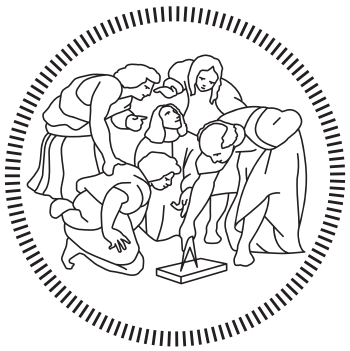
Politecnico di Milano
Scuola di Ingegneria Aerospaziale e dell'Informazione
Corso di Laurea Magistrale in Space Engineering



Assessment of Deep Reinforcement Learning for flexibility enhancement of planetary landing guidance and control

Reinforcement learning of policies for reusable launchers planetary landing

Advisor: Prof. Michele Lavagna
Co-Advisor: Dr. Robert Hinz
Eng. Andrea Brandonisio

Thesis by:
Davide Iafrate Matr. 962657

Academic Year 2021–2022

Maktub...

Acknowledgments

This work is the outcome of 5 years of personal and professional growth. Every person I've encountered on this journey helped me become who I am today, even unknowingly. A book might barely be enough to fit them all in, one page is definitely not.

I want to thank my family, in particular my parents for helping me pursue my goals and dreams, and my brother whom I wish to be able to do the same.

I thank all the friends that helped me learn, grow and endure, your help was priceless and never to be forgotten.

I thank my childhood friends Andrea and Francesco for sharing with me this rollercoaster ride and making it fun in the meanwhile.

I thank my friend Vincenzo, who taught me how to learn and what means to withstand challenges and thrive in them.

Abstract

The planetary landing problem is gaining relevance in the space sector, spanning a wide range of applications from unmanned probes landing on other planetary bodies to reusable first and second stages of launcher vehicles. It is therefore crucial to assess the performance of novel techniques and their advantages and disadvantages.

The purpose of this work is the development of an integrated 6DOF guidance and control approach based on reinforcement learning of deep neural network policies for fuel-optimal planetary landing control, specifically with application to a launcher first stage terminal landing, and the assessment of its performance and robustness.

3DOF and 6DOF simulators are developed and encapsulated in MDP-like (Markov Decision Process) industry-standard compatible environments. Particular care is given in thoroughly shaping reward functions capable of achieving the landing both successfully and in a fuel-optimal manner. A cloud pipeline to effectively train an agent using a PPO reinforcement learning algorithm to successfully achieve the landing goal is developed and the performance and robustness of the obtained policy is assessed in an industrially-validated 6DOF simulator in the presence of additional disturbances and uncertainties in the model parameters.

Sommario

Il problema dell'atterragio planetario sta assumendo un ruolo sempre più centrale nel settore spaziale, attraverso un ampio spettro di applicazioni, da sonde autonome che atterrano su altri corpi celesti a primi e secondi stadi di lanciatori riutilizzabili. E' quindi essenziale valutare le prestazioni di nuove tecniche e i loro vantaggi e svantaggi in queste applicazioni.

Lo scopo di questo lavoro è lo sviluppo di un approccio integrato di guida e controllo a 6DOF basato sull'apprendimento di policy basate su deep neural networks tramite reinforcement learning per ottenere traiettorie di atterraggio planetario ottimale che minimizzino il consumo di carburante. In particolare è studiata l'applicazione alla fase terminale di atterraggio del primo stadio di un lanciatore e la valutazione delle sue prestazioni e della sua robustezza.

Due simulatori a 3DOF e 6DOF sono sviluppati e incapsulati in environment basati sul concetto di MDP (Markov Decision Process), compatibili con gli standard industriali. Particolare attenzione è stata dedicata al modellare accuratamente funzioni di reward in grado di realizzare l'atterraggio con successo e in modo ottimale dal punto di vista del carburante. Una pipeline cloud per addestrare efficacemente un agente che utilizza l'algoritmo di reinforcement learning PPO per raggiungere con successo l'obiettivo di atterraggio è sviluppata e le prestazioni e la robustezza della policy ottenuta sono valutate in un simulatore 6DOF validato a livello industriale in presenza di disturbi aggiuntivi e incertezze parametriche del modello.

Contents

List of Figures

List of Tables

Nomenclature

$\mathcal{F}_B$	Body-fixed Reference Frame
$\mathcal{F}_I$	Inertial Reference Frame
$\oslash$	Hadamard division
$\boldsymbol{u}$	Control input vector
$\boldsymbol{x}$	State vector
$\boldsymbol{y}$	State vector (3DOF environment)
H_∞	H-infinity synthesis
3DOF	3 degrees of freedom
6DOF	6 degrees of freedom
CoM	Center of Mass
CoP	Center of Pressure
COTS	commercial off the shelf
IC	Initial Conditions
ISA	International Standard atmosphere
MPC	Model Predictive Control
OCP	Optimal Control Problem
ODE	Ordinary Differential Equations
PID	Proportional Integrative Derivative controller
CD	Continuous Deployment
VM	Virtual Machine

$[\psi, \theta, \phi]_{\text{lim}}$ Limit Yaw, Pitch and Roll angles

DRL Deep Reinforcement Learning

α Learning rate

$\boldsymbol{\theta}$ Network parameters

ϵ Clipping parameter

$\mathbb{E}$ Expected value

$\mathcal{A}$ Action Space

$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}_k)$ surrogate advantage function

$\mathcal{O}$ Observation Space

π Policy

π^* Optimal policy

τ Trajectories (also rollouts, episodes)

$\boldsymbol{a}_{targ}$ Target acceleration

$\boldsymbol{s}_{t_k}$ Environment state at time t_k

$A^{\pi_{\boldsymbol{\theta}_k}}(\boldsymbol{s}, \boldsymbol{a})$ Advantage function

D_{KL} Kullback–Leibler divergence

$J(\pi)$ Expected return over a trajectory

$Q^*(\boldsymbol{s})$ Optimal Action-Value function

$Q^{\pi}(s)$ Action-Value function

$R(\tau)$ non-discounted finite-horizon return

$R(\tau)^{\text{inf}}$ discounted infinite-horizon return

$R(\boldsymbol{s'}, \boldsymbol{s}, \boldsymbol{a})$ Reward function

r_t Reward at time t

$ReLU$ Rectified Linear Unit

S Entropy function

$tanh$ Hyperbolic tangent

$V^*(\boldsymbol{s})$ Optimal value function

$V^\pi(s)$ Value function

v_{targ} Target velocity

DDPG Deep Deterministic Policy Gradient

DNN Deep Neural Networks

ELM Extreme Learning Machines

GPU Graphic Processing Unit

MDP Markov Decision Process

POMDP Partially Observable Markov Decision Process

PPO Proximal Policy Optimization

RL Reinforcement Learning

SAC Soft Actor-Critic

TRPO Trust-Region Policy Optimization

ZEM Zero-Effort Miss

ZEV Zero-Effort Velocity

Chapter 1

Introduction

In this thesis an analysis of the feasibility of utilizing Reinforcement Learning (RL) techniques for planetary landing is carried out. Emphasis is placed on the case of reusable first stages of launch vehicles, as the work is developed in the context of an internship within Deimos Space [1], a commercial company carrying out several projects on this type of vehicles.

1.1 Motivation

The landing problem consists of achieving a successful touchdown on a planetary body within a prescribed location and velocity with a certain attitude (usually upright) and null terminal angular velocity. This goal can be optimized in terms of time, propellant consumption and/or terminal errors' minimization, depending on the needs. This problem has been rising in relevance in the recent years, in an effort to make space access more economically accessible by means of reusable launcher vehicles. It is also relevant for the future planetary exploration goals of space agencies, to have reliable, precise and efficient landings close to scientifically relevant sites.

Through the years several techniques have been developed to address it, starting from simple fixed-guidance solutions to advanced optimization techniques. The first applications were the lunar landings during the first space race: one of the most iconic solutions was the polynomial trajectory guidance for the Apollo missions. This consisted in describing the trajectory using polynomials split through different phases of the landing profile [2], a solution adopted due to the limited computational resources available.

To improve the fuel efficiency of the trajectory, optimal *guidance* laws have emerged [3] for the case of point-mass dynamics, lacking however equally optimal solutions in the atmospheric case and still needing a mapping to actuator actions through a tracking controller.

These methods however lack the ability to optimize more complex dynamics

and to enforce constraints. Recently, substantial gains in efficiency and flexibility in the ability to handle constraints have been made by approaching the problem with Second Order Cone Programming (SOCP) methods [4],[5]. These methods manage to obtain optimal guidance solutions in the presence of linear dynamics and convex inequality constraints. These methods exploit the fact that while *non-convex* functions can have multiple local minima, *convex* functions have no more than one minimum, making optimization much easier: they work by *convexifying* non-convex constraints of the state and control inputs. They solve the convex *relaxed* problem associated to the non-convex one by *convexifying* the non-convex constraints. It has been proven that the optimal solution of the relaxed problem coincides with the optimal solution of the non-convex one [6]. The great advantage of this method is that it can compute a solution in bounded polynomial time, making it suited to real-time applications. This approach is limited by the fact that the dynamics that can be optimized must be linear and not all constraints can be convexified. Furthermore, conical glideslope constraints are usually used, and they might not be appropriate for trajectories which need to avoid steep obstacles close to the landing pad. This approach can be applied to 3DOF guidance problems, meaning that only the translational motion of the lander is optimized, while the attitude needs to be controlled separately to track the required control force: this is necessary as typical launcher vehicles are under actuated, as the thrust vector is constrained to lie within a certain region by the attitude of the system. An example of application of this method is the G-FOLD (Guidance for Fuel Optimal Large Divert) algorithm developed at JPL [7], of which a real-time implementation has been flight-tested on actual demonstrator vehicles by Masten Aerospace.

To overcome some of these limitations, *successive convexification (SCVX)* methods iteratively optimize convexified problems for guidance computation to generate 6DOF optimal open-loop trajectories, such as [8],[9]. At each iteration the convexification is repeated using as initial guess the solution from the previous time step, until convergence is reached. This approach requires an additional controller to track the required thrust components using the thruster or a Model Predictive Control approach re-optimizing the control sequence asthe trajectory is recomputed at each time step, significantly increasing the computational burden.

To add a guarantee of stability and performance of the controller, robust control techniques have been used in the European space sector, usually in the form of H_∞ synthesis (either structured or unstructured), such as in [10],[11]. These methods typically use cascade PID controllers, as in 1.1: an outer loop controller receives a reference trajectory to track from a guidance algorithm; it then outputs reference attitude angles to feed an inner loop controller controlling the actuators. The performance of these methods however can suffer when dealing with time-varying dynamical systems.

At present time the state-of-the-art approaches split the process in sequential steps, first computing the guidance from an estimate of the state given by the

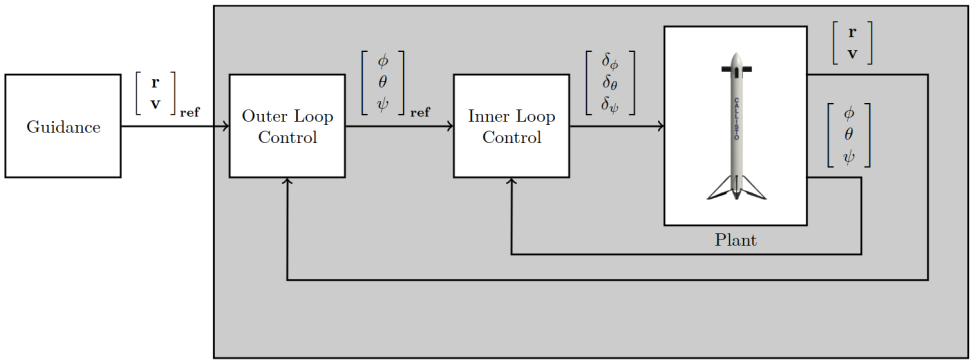


Figure 1.1: The inner and outer loop controller architecture applied to the aerodynamic descent phase of the CALLISTO ESA reusable launcher demonstrator [10]. The control inputs to the plant $(\delta_\phi, \delta_\theta, \delta_\xi)$ are *virtual deflections* mapping to the fins control angles through aerodynamic coefficients

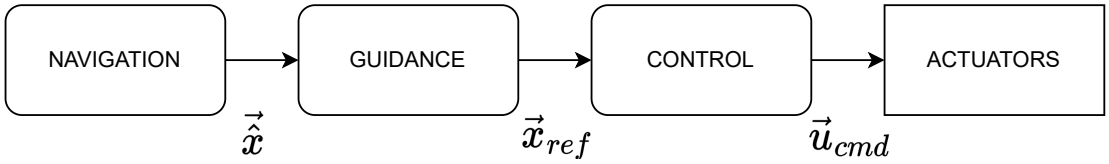


Figure 1.2: Structure of classical landing architectures: the navigation system produces from sensors an estimate of the state $\vec{\hat{x}}$, which the guidance block turns in a commanded trajectory $\vec{x}_{ref}$. Finally, the control block transforms this in actuator commands $\vec{u}_{cmd}$. Each block serves a specific function, making their individual design easier but causing performance to suffer.

navigation subsystem and then tracking it with a controller, as shown in Fig.1.2.

This has several limitations, such as a difficulty in designing separately the guidance and control blocks, the guidance block needing to ensure feasibility of the generated trajectory and the possibility of the control being suboptimal and incurring in saturations. Also, they could restrict the type of possible maneuvers performed. In the present work it is assumed that the navigation subsystem is capable of giving an accurate estimate of the state of the lander (which is reasonable in the case of Earth landings) and the focus will be on Guidance and Control.

1.1.1 Learning-based techniques for landing G&C

All these limitations of classical G&C architectures spur the question if an integrated approach to G&C design is possible and what level of performance (in terms of both objective function minimization and robustness of the solution) is

currently achievable. The limitations and range of applicability of the developed solution also need to be assessed in order to inform adequately the G&C designers. A promising answer for this question makes use of Artificial Intelligence learning techniques to develop integrated Guidance and Control solutions.

As reported in [12] there have been several studies in the last decade applying artificial intelligence and reinforcement learning to spacecrafts' GNC problems.

Neural networks are used to approximate parameters of a controller with fixed structure such as PID gains, tuning matrices for LQR or other types of parameters. The base controller can be developed through any standard control technique, such as PID, LQR, using Lyapunov control methods [13] or ZEM/ZEV controllers where a network learns to output the optimal controller parameters using reinforcement learning [14]. An interesting subcategory of this approach are extreme-learning machines (ELM), neural networks with a single hidden layer trained very quickly using an inverse parameters' matrix rather than Stochastic Gradient Descent [15] [16]. Similar approaches are also used for guidance tasks [17], also with optimal control techniques using neural networks to generate good initial guesses [18], [19].

Optimal Control solutions using Deep Neural Networks[20] address the high computational burden of solving Optimal Control guidance problems in real-time applications: to seek a feasible real-time solution a Deep Neural Network (DNN) is trained using a large database of previously computed optimal solutions, to obtain an approximator for the optimal solution of the problem. This means shifting the computational burden to the *offline training* of the network. This results in a quasi-optimal guidance law which can then be deployed easily on available commercial off-the-shelf hardware (COTS) achieving a real-time performance. It also addresses the need for a good initial guess to have convergence, as there could be non-converging scenarios which would not output a control action: the neural network allows to have a deterministic, guaranteed control output at each time step.

Several studies use adaptive controllers which are either built from or expanded with [21] neural networks. For example in [22] (in the context of sole attitude control of a space station) the synthesis of a nonlinear robust controller in the form of neural networks is performed and the parameters of the neuro-controller are adaptively modified to account for a time-varying inertia matrix.

Other approaches yet explore using learned dynamics, training a neural network from experimental data to obtain a quickly differentiable representation of nonlinear dynamics, which can then be employed with more classical optimal control techniques such as Model Predictive Control [23]. This allows for unconstrained model selection and GPU acceleration during optimization.

Reinforcement learning has also been studied in several fashions, using supervised learning to obtain the initial approximation for the neural network subsequently optimized with reinforcement learning methods [24] or through reinforcement meta-learning[25], where the agent learns to act effectively not just in one

task, but in a series of related tasks: in this way when facing a new task that has never been encountered during training, the agent will still be familiar with the situation and will be able to learn faster.

Deep Reinforcement Learning for direct control seeks to find a *control policy* directly mapping the state of the system (or a partial observation of the state) to a control action. This approach is highly flexible as it can take into account actuator limitations and unmodeled dynamics while learning the mapping from state observation to actuator action. This approach does not require pre-collected data (either from simulations or real-data) but rather optimizes the policy (represented with a Deep Neural Network) by simulating the behavior of the system and collecting information on the reward obtained at each time step. The reward is a scalar number given by a reward function, which measures how close we are to achieving an arbitrary objective. Using this information to *learn* the optimal policy, mapping actions to take to the different states of the lander. Reinforcement Learning makes this approach extremely versatile as the control function can learn its own structure, not having to fit any predefined one. There are several variations of this technique, but it can mainly be subdivided in *model-based* and *model-free*: the first uses a model of the system to guide learning of the controller while the second doesn't. Previous applications to planetary landing problems [26] show that it suits this kind of G&C problems.

Ultimately the decision was made to focus on Deep Reinforcement Learning from state observations, as it allows maximum flexibility for representation of the dynamics: the *model-free* approach has been chosen due to the increased flexibility in modeling the nonlinear dynamics and its higher effectiveness and versatility. Furthermore, there exists a gap in the application to launcher's first-stage landing scenario on Earth: previous studies [24], [14],[26] that approached the landing problem using this technique focused on applications to other planetary bodies rather than Earth, with less significant lower atmospheric effects.

In previous works[26], the focus was placed on narrowing down the landing location from a wide spatial distribution of initial conditions, with limited dispersion on the initial mass. The existing works analyze cases where the command vector is the force components to be generated by the actuator or the individual thrust for each one of several fixed thrusters: no work employs direct control of gimbaled thruster, which significantly changes the behavior of the lander. The limitations of these approaches are in the long convergence times, sometimes of about a week of training, which strongly limits the ability to iterate quickly on the shape of the reward function informing the algorithm about the effectiveness of the obtained trajectory.

1.2 Objectives of the thesis

In this work an assessment of the maturity of reinforcement learning (RL) techniques for development of integrated Guidance and Control policies is conducted. The advantage compared to classical learning techniques is the possibility of higher performance thanks to the integrated G&C approach, an increased flexibility in the definition of goals and constraints, and the possibility of easily expanding or modifying the actuators' architecture. We aim to address the problem of long run times for this type of algorithms to improve on the speed of design iteration. Quantifying the robustness of the obtained control policy is also important, in particular in the case of unmodeled dynamics and/or uncertainty on the parameters. Thus, the research objectives that we seek to address with this thesis are:

- Applying model-free reinforcement learning to consistently achieve successful atmospheric landing by directly controlling the launcher's actuators and assessing its robustness to unmodeled dynamics.
- Developing a validated 6DOF simulation environment, expandable with more actuators and including the relevant dynamics for atmospheric planetary landing.
- Developing a training software pipeline that enables fast iteration on the design of the integrated guidance and control policy.

The first step was the development of the simulator and environment, to be used by the model-free Reinforcement Learning algorithm to learn a control policy. The training software pipeline development was required due to the long training times of the algorithm which would have prevented testing rapidly the changes to the environment and the algorithm.

1.3 Structure of the thesis

This document is organized as follows:

- **Chap.1 Introduction:** an overview of the current state of the art for planetary landing G&C is given, showing the classical architecture employed. An overview on the applications of learning techniques in aerospace G&C is presented and the selection of Reinforcement Learning is motivated. Finally a recap of the objectives of the thesis is shown.
- **Chap.2 Reinforcement Learning description:** a brief survey of machine learning techniques for control is provided and, after motivating the choice of Reinforcement Learning, an overview of the most used techniques is provided. Finally, the used algorithm (PPO) is explained and its choice motivated.

- **Chap.3 Overview of the landing problem:** the dynamics of the problem are modeled, for both a planar 3DOF, and a full 6DOF case. The landing problem is formulated and cast in a form suited to reinforcement learning algorithms.
- **Chap.4 Environment definition:** the dynamics are wrapped in an environment suited to RL and its structure is detailed. A thorough explanation of how the constraints and goals are enforced through the reward function is given. Finally, a description of the software setup used is provided and validation of the developed simulator is highlighted.
- **Chap.5 Results for 3DOF environment:** the results obtained in terms of performance of the algorithm are shown and a trade-off study across different activation functions is carried out.
- **Chap.6 Results for 6DOF environment:** the results in a realistic environment are presented. Monte Carlo analysis on the nominal environment and a sensitivity analysis on an off-nominal environment (simulating previously unmodeled dynamics and uncertainties) are carried out to assess the robustness of the obtained policy to unmodeled effects.

The problem has been approached through subsequently more general cases, to gain deeper understanding of the interaction between the change in the reward function used, the choice of hyperparameters and the convergence of the algorithm. We have first addressed a 3DOF case with simplified initial conditions (lower initial velocity and height, see Table 4.4), then moving to the 6DOF environment as in Table 4.5. For this, two Python simulators were developed, one for the 3DOF problem and one for the 6DOF case. The second was validated using a previously validated 6DOF simulator. After satisfactory results were obtained, we moved to realistic initial conditions taken from the flight profile of a Falcon 9, again first solving the problem in 3DOF (initial conditions in Table 4.6) and then moving to 6DOF (see Table 4.7 for details). These steps are shown in order in Fig.1.3. The whole training phase was greatly facilitated by the software pipeline developed, that allowed easily training on Cloud Virtual Machines and monitoring the training metrics through an interactive web interface.

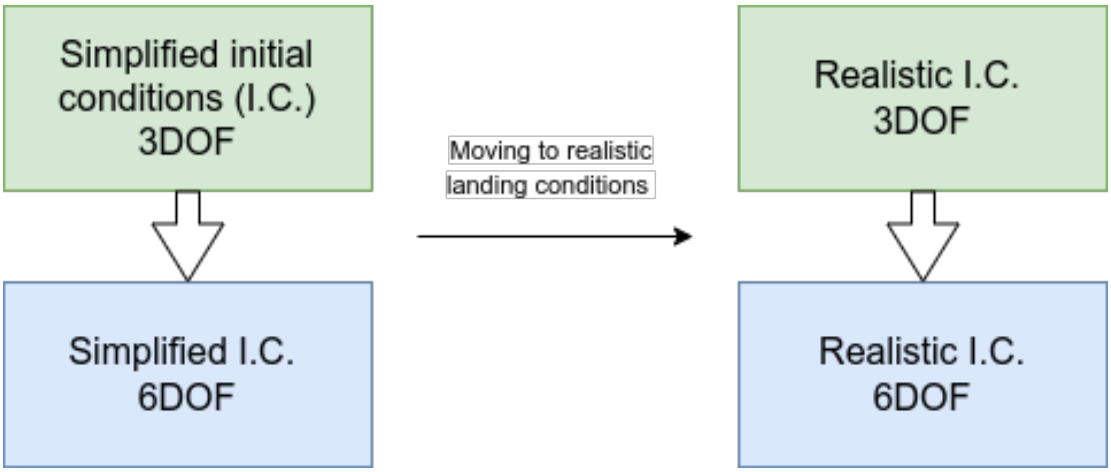


Figure 1.3: Steps in the development process: we first started from simplified initial conditions for the landing going from a 3DOF case to a 6DOF one; then we moved to realistic initial conditions from Falcon 9 landing data, again moving from the 3DOF case to the 6DOF one.

Chapter 2

Reinforcement learning

Reinforcement Learning (RL) is a set of techniques through which it is possible to learn policies to control the behavior of an agent, attempting to maximize the amount of reward obtained from a certain environment. In this chapter we show several classification domains for the different RL algorithms, explaining mathematically and motivating the selection of the *Proximal Policy Optimization* (PPO) algorithm.

2.1 Brief introduction to reinforcement learning

The environment is modeled as a Markov Decision Process (MDP) in case the *state* of the system (i.e. all the variables controlling the system) can be observed or a Partially Observable Markov Decision Process (POMDP), in which only an *observation* of the state of the system is available. In the following the formulation for MDPs is described and the terms state and observation are used interchangeably. There are several possible techniques to solve the reinforcement learning problem (Dynamic Programming, tabular methods, Neural Networks etc. [27], [28]) and in this work Deep Neural Networks (i.e. neural nets with more than 1 hidden layer) are used to tackle this problem.

2.1.1 Elements of Reinforcement Learning

There are several elements which make up the reinforcement learning [29]:

- **action space $\mathcal{A}$**: which is the set of all possible actions $\boldsymbol{a}$ of the agent and can be discrete if it has a finite number of actions or continuous.

- **Observation space $\mathcal{O}$ (and state space $\mathcal{S}$)**: is the set of all possible observations $\boldsymbol{o}$ of the system state $\boldsymbol{s}$ and can again be discrete or continuous.

- **Policy**: the policy π is a function controlling the behavior of the agent, that is giving an action based on an observation received from the environment. Since the policy controls the behavior of the agent often the two terms are used to mean policy. In deep reinforcement learning the policy is parametrized using a deep neural network, meaning that the weights and biases of the neural network are used to parametrize it and is denoted as $\pi_{\boldsymbol{\theta}}$ to indicate that the parameter vector $\boldsymbol{\theta}$ is used.

- **Trajectories** (also called rollouts, episodes): they are a sequence of state and action pairs Eq.2.1.

$$\tau = (\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{s}_1, \boldsymbol{a}_1, ...) \tag{2.1}$$

  The first state is sampled from an initial state distribution ρ_0.

- **State transition function**: the sequence of states is determined by the state transition function which returns the successive state based on the current state and the action taken (either in a deterministic or probabilistic sense) as Eq.2.2.

$$\boldsymbol{s}_{t+1} = \boldsymbol{s}' = f(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{2.2}$$

- **reward and returns**: at each transition a scalar reward is given to the agent based on the state of the environment, the next state and the action taken, according to a *reward function*2.3.

$$r_t = R(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') \tag{2.3}$$

  The goal of the agent is to maximize the amount of reward collected along a trajectory, calculated as either the *non-discounted finite-horizon return*2.4,

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.4}$$

  or the *discounted infinite-horizon return*2.5,

$$R(\tau)^{\text{inf}} = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.5}$$

  which computes the return discounting the value of future rewards, which both guarantee convergence of the sum and can be tweaked to emphasize obtaining a reward closer in time.

In order for the algorithm to solve the problem it is useful to understand the *value* that a certain state has or the value that a state-action pair has. To do these two types of *value functions* are employed:

- **Value function**2.6: it measures the value that a specific state possesses as the expected return starting in the state and following the policy π.

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|\boldsymbol{s}_0 = \boldsymbol{s}\right]. \tag{2.6}$$

  This function has a respective *optimal value function*2.7

$$V^{*}(\boldsymbol{s}) = \max_{\pi} V^{\pi} \tag{2.7}$$

- **Action-Value function**2.8: gives the expected return being in a state $\boldsymbol{s}$, taking an arbitrary action $\boldsymbol{a}$ and then following the current policy

$$Q^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|\boldsymbol{s}_0 = \boldsymbol{s}, \boldsymbol{a}_0 = \boldsymbol{a}\right]. \tag{2.8}$$

  Again there is an *optimal action-value function*, shown in Eq.2.9.

$$Q^{*}(\boldsymbol{s}) = \max_{\pi} Q^{\pi} \tag{2.9}$$

The key problem of reinforcement learning is to maximize the expected return of the agent over a trajectory, that is the cost function in Eq.2.10,

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right] \tag{2.10}$$

thus retrieving the optimal policy π^{*} as Eq.2.11

$$\pi^{*} = arg\max_{\pi} J(\pi) \tag{2.11}$$

2.2 Reinforcement Learning algorithms classification

There are several ways that RL can be employed to learn how to map control actions to system observations (the so-called *control policy* π) to accomplish a certain task. The first distinction has to be done between **model-based** and **model-free** reinforcement learning. The first has either access or learns a baseline model of the environment and this allows the algorithm to use the model to plan and make predictions, allowing for an improvement in *sample efficiency*, meaning that the algorithm requires less exploration to optimize the policy. However, there is the downside that these algorithms actually require to have a model of the environment (or to learn it) which might be too complex to achieve and thus the *model-free* algorithms allow learning a policy without the need of modeling the environment and generally are easier to implement and to tune, thus these are the ones most used. The two main categories of *model-free* RL problems are the **policy-based** methods and the **value-based** methods.

- **Q function-based methods** Q function-based methods seek to estimate the *state-action value function* giving the estimated future cumulative discounted reward given that a certain *state* s is reached. The policy is then retrieved by picking the actions that maximize the state-action value function as Eq.2.12.

$$\pi = \arg\max_a Q(s, a) \tag{2.12}$$

- **Policy-Based methods** are seeking to directly find an optimal control policy $\pi^*(t)$ maximizing the rewards obtained. The policy is usually parametrized according to some parameters $\boldsymbol{\theta}$ (for example if we are using a neural network the parameters would be the weights and biases of the network) and then iteratively optimized by seeking the parameters that maximize the cost function J by following the direction of the gradient of this function with respect to the parameters as in Eq.2.13.

$$\boldsymbol{\theta} \rightarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.13}$$

- **Mixed methods** are methods mixing both *model-free* and *model-based* techniques. The so-called *imagination-augmented* [30] [31] method combines model-based and model-free RL, learning to interpret environment models to *augment model-free decisions*. This allows for flexibility in the development of the DNNs as they are decoupled and "makes use" of the fact that the environment is constrained by its dynamics. The dynamic of the system is modeled through ODEs and employed in the model-based part of the controller to estimate (i.e. '*imagine*') future states. Even if it doesn't take into account stochastic events or the full dynamics of the launcher this model can still aid in the generation of the control action by 'filtering' the model-free actions, in a way akin to MPC.

Another way to characterize RL algorithms is whether they learn only applying the current policy or not, if they are respectively *on-policy* or *off-policy*:

- **On-policy** algorithms learn value functions using information obtained by exploiting the current policy to explore the environment. An example of a modern on-policy algorithm is Proximal Policy Optimization (PPO) [32].

- **Off-policy** These algorithms use observations obtained by previous policies or an expert ('experience replay'). One of the most used off-policy solutions are the Soft Actor-Critic (SAC) algorithm [33] and the Deep Deterministic Policy Gradient (DDPG) [34].

The main trade-off between the two methods is stability (i.e. convergence to a solution in reasonable time) vs sample efficiency (i.e. making use of as much data obtained as possible). The first is usually better for the on-policy algorithms, and consequently also for policy-based algorithms that are optimizing *directly* the policy as they are following a gradient ascent on a cost function, while the latter is higher in the off-policy algorithms, due to the ability to re-use previous transitions.

A recap of the classification is shown in 2.1.

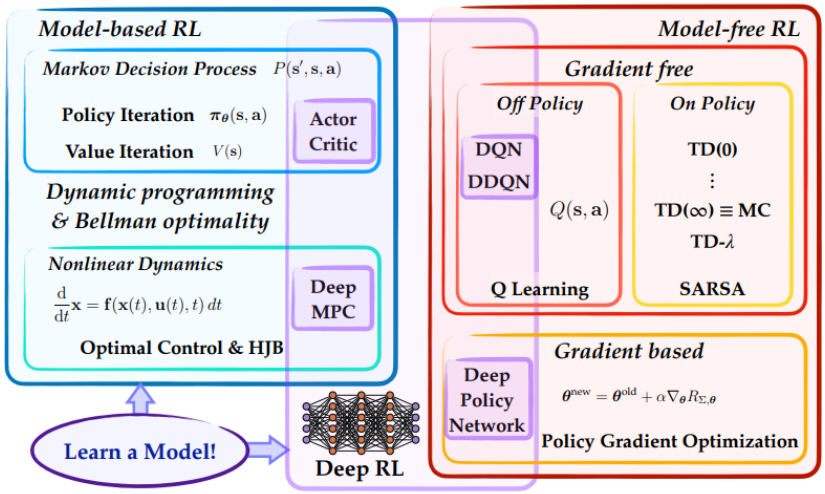


Figure 2.1: A recap of the classification of Reinforcement Learning domains. The lines between them are not strict and there are techniques which overlap over several domains [27]

2.3 The Proximal Policy Optimization algorithm

The Reinforcement Learning algorithm chosen to tackle the landing problem is the *Proximal Policy Optimization* (PPO) algorithm, a type of on-policy, gradient-based algorithm characterized by a high learning efficiency.

2.3.1 Motivation for choice

The PPO algorithm has been chosen due to his versatility and wall-clock performance. It has been applied in the literature to many environments, demonstrating state-of-the-art performance in many of these applications, with minimal hyperparameters tuning required [35]. This aspect is critical in the applied scenario as convergence of the policy requires millions of episodes and significant wall-clock time which make hyperparameters tuning infeasible due to limitations in the computational resources available.

If more resources were available it would become feasible to train several policies in parallel, sweeping over a broader range of hyperparameters to optimize performance of the algorithm. Indeed, one way to do so could be performing the hyperparameters sweep in the 3DOF environment and then use these optimized hyperparameters to train the policy in the 6DOF environment similarly as how tuning of the reward function has been performed (see Sec. 4.5).

Another reason for the selection is that it is an on-policy algorithm, iterating and optimizing directly the policy network which result in a training that is more stable and reliable. Indeed, the training stability is a critical limitation to the exploitation of the designed reward function, as having an excessively unstable training may result in not succeeding in finding the terminal reward. Training instability could also result in not being able to exploit the reward information due to diluting it too much with other episodes' trajectories, thus not having a strong enough learning signal to make the policy parameters converge to the optimal ones.

2.3.2 The Trust Region Policy Optimization algorithm

The PPO family of on-policy algorithms are an evolution of the *Trust Region Policy Optimization* (TRPO) which ease the computational burden of each update step to speed up learning of the optimal policy. The TRPO algorithm uses a constrained approach for the maximization of the advantage function Eq.2.14.

$$A^{\pi_{\boldsymbol{\theta}_k}}(\boldsymbol{s}, \boldsymbol{a}) = Q^{\pi_{\boldsymbol{\theta}_k}}(\boldsymbol{s}, \boldsymbol{a}) - V^{\pi_{\boldsymbol{\theta}_k}}(\boldsymbol{s}) \tag{2.14}$$

This function measures how much better a given action is compared to taking an action according to the current policy $\pi_{\boldsymbol{\theta}_k}$. The constraint limits the change in policy at each update step. This can be written mathematical as the maximization of the *surrogate advantage function* $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}_k)$, where $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_k$ are respectively the generic and previous parameters of the parametrized policy network $\pi_{\boldsymbol{\theta}}$ at a given policy update iteration, subject to a constraint on the average KL-divergence of the policies $\bar{D}_{KL}(\boldsymbol{\theta}||\boldsymbol{\theta}_k)$, as shown in equation 2.3.2.

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}_k) \tag{2.15}$$

$$s.t.\, \bar{D}_{KL}(\boldsymbol{\theta}||\boldsymbol{\theta}_k) \leq \delta \tag{2.16}$$

In this equation the *surrogate advantage function* $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}_k)$2.17 measures how the generic policy $\pi_{\boldsymbol{\theta}}$ performs compared to the previous policy $\pi_{\boldsymbol{\theta}_k}$, weighted by how much they differ and is computed as the expected value over the states and actions (obtained through rolling out the previous policy $\pi_{\boldsymbol{\theta}_k}$) of the ratio between the probabilities of the generic and previous policy $r_k = \dfrac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)}$ multiplied by the

previous estimate of the advantage function $A^{\pi_{\boldsymbol{\theta}_k}}(s,a)$:

$$\mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) = E_{s,a\sim\pi_{\boldsymbol{\theta}_k}}\left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} A^{\pi_{\boldsymbol{\theta}_k}}(s,a)\right] \tag{2.17}$$

The average KL-divergence of the two policies across the states visited under the old policy $\pi_{\boldsymbol{\theta}_k}$ is shown in Eq.2.18.

$$\bar{D}_{KL}(\boldsymbol{\theta}||\boldsymbol{\theta}_k) = E_{s\sim\pi_{\boldsymbol{\theta}_k}}\left[D_{KL}(\pi_{\boldsymbol{\theta}}(\cdot|s)||\pi_{\boldsymbol{\theta}_k}(\cdot|s))\right] \tag{2.18}$$

where the KL-divergence measures how different the two policies are and is computed as in Eq.2.19.

$$D_{KL}(\pi||\pi_{\boldsymbol{\theta}_k}) = \mathbb{E}\left[\log\left(\frac{\pi}{\pi_{\boldsymbol{\theta}_k}}\right)\right] \tag{2.19}$$

This maximization problem is solved through a second-order optimization method which requires computing the inverse of the hessian matrix of the network, which can have millions of parameters. This computation is very expensive so the PPO algorithm tries to enforce the constraint in the KL-divergence of the policies through a clipped objective function, turning the constrained optimization problem into an unconstrained one which can be solved through first-order method, resulting in a much lower computational burden.

2.3.3 Algorithm description

The PPO algorithm avoids having to solve the second-order maximization problem of TRPO while constraining the new policy to not be too far from the old one. There are two ways in which this can be achieved[29]:

- **loss penalty term**: a negative penalty term that grows (in absolute value) with the increase in KL-divergence is added to the loss function that is maximized, automatically adjusting the weight of this additional term at each optimization iteration
- **loss clipping**: the loss function is clipped to incentivize the difference between policy update steps to be within a certain range

The first approach has two steps at each policy iteration:

- Using gradient descent, optimize the *penalized objective*2.20,

$$L_{penalty} = \mathbb{E}\left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)}\hat{A} - \beta\bar{D}_{KL}(\boldsymbol{\theta}||\boldsymbol{\theta}_k)\right] \tag{2.20}$$

- compute the expected value of the divergence $d = \mathbb{E}[\bar{D}_{KL}(\boldsymbol{\theta}||\boldsymbol{\theta}_k)]$ and update the coefficient β in Eq.2.20 as (this coefficient controls how aggressive the policy update step is).

$$\begin{cases} \beta \leftarrow \beta/2 & \text{if } d < d_{\text{targ}}/1.5 \\ \beta \leftarrow \beta/2 & \text{if } d > d_{\text{targ}} \times 1.5 \end{cases} \tag{2.21}$$

The second approach is the one commonly used as it is simpler and has similar results. The policy is updated, as in TRPO, as:

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s, a, \boldsymbol{\theta}_k, \boldsymbol{\theta}) \right], \tag{2.22}$$

however now the loss function is computed as in Eq.2.23, being r_k Eq.2.24 the probability ratio between previous iteration and generic policies.

$$L(s, a, \boldsymbol{\theta}_k, \boldsymbol{\theta}) = \min \left(r_k A^{\pi_{\theta_k}}(s, a), \;\; \mathrm{clip}\left(r_k, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a) \right) \tag{2.23}$$

$$r_k = \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} \tag{2.24}$$

The clipping behavior is regulated by the ϵ parameter, usually selected as a small value $\epsilon \simeq 0.1 \div 0.3$. The algorithm can be summarized as in the following pseudocode:

Algorithm 1 Pseudocode of the PPO-Clip algorithm, from OpenAI Implementation [29]

1: Input: initial policy parameters $\boldsymbol{\theta}_0$, initial value function parameters $\boldsymbol{\phi}_0$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\} = \{\langle \boldsymbol{s}_i, \boldsymbol{a}_i \rangle, \}$ by running policy $\pi_k = \pi(\boldsymbol{\theta}_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t = \sum_{t'=t}^{T} R(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}, \boldsymbol{s}_{t'+1})$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(r_k A^{\pi_{\theta_k}}(\boldsymbol{s}, \boldsymbol{a}), \;\; \mathrm{clip}\left(r_k, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(\boldsymbol{s}, \boldsymbol{a}) \right), \tag{2.25}$$

via stochastic gradient ascent with Adam optimizer.
7: Fit value function by regression on mean-squared error:

$$\boldsymbol{\phi}_{k+1} = \arg\min_{\boldsymbol{\phi}} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_{\phi}(s_t) - \hat{R}_t \right)^2, \tag{2.26}$$

typically via gradient descent
8: **end for**

There is also the possibility of augmenting the loss function with an *entropy bonus*, a term that increases with the entropy of the policy. This term makes

the loss function bigger when the policy has a very narrow distribution, that is when one action is much more likely than the others. This increase in the loss function tries to avoid premature convergence of the policy to a local optimum and is shown in Eq.2.27.

$$L^{CLIP+S}(s, a, \boldsymbol{\theta}_k, \boldsymbol{\theta}) = \mathbb{E}\left[L(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{\theta}_k, \boldsymbol{\theta}) + c_{\text{entropy}} S[\pi_{\boldsymbol{\theta}}(\boldsymbol{s})]\right] \tag{2.27}$$

The entropy $S[\pi_{\boldsymbol{\theta}}(\boldsymbol{s})]$ is multiplied by the coefficient c_{entropy} and is computed as in Eq.2.28.

$$S[\pi_{\boldsymbol{\theta}}(\boldsymbol{s})] = \mathbb{E}[-\log \pi_{\boldsymbol{\theta}}(\boldsymbol{s})] \tag{2.28}$$

2.3.4 Neural Network structures

The neural networks are used in the PPO algorithm to estimate the policy function and the value function. The network structure employed is a MultiLayer Perceptron (MLP) and it is different depending on which of the two sets of initial conditions is used (See Fig.1.3 for an overview on the two sets). This is due to the need to learn a more complex policy in the case of *realistic initial conditions* compared to the simplified ones, thus needing a network with more parameters. The network is made by an input layer for the observation, two hidden layers with sizes shown in Table 2.1 and an output layer for the action or the value, similar to the one shown in Fig.2.2. Both value and policy network have this structure, with the difference being the last layer, which outputs a single scalar value in the value network case.

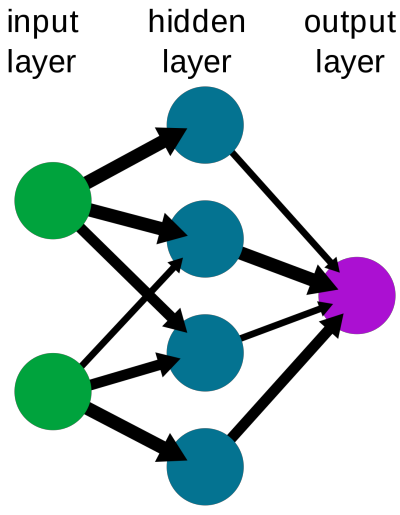


Figure 2.2: Structure of the neural network (thicker arrows represent larger weights for that link).

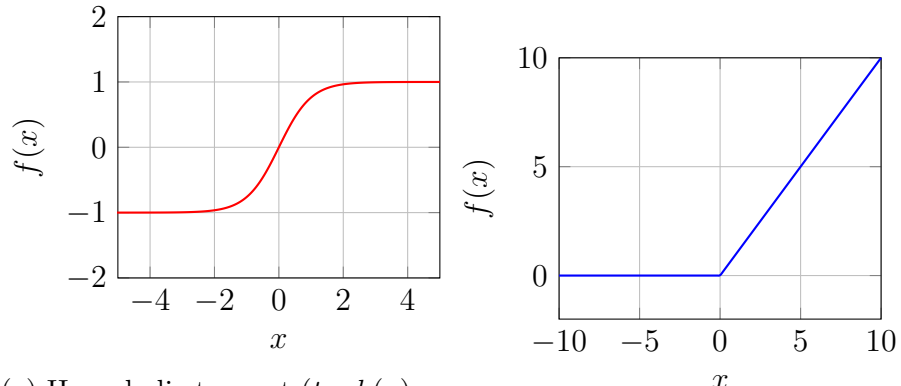
	Simplified IC	Realistic IC
Hidden layer 1	64	128
Hidden layer 2	64	64

Table 2.1: Structures of the used networks: the more challenging realistic initial conditions require a bigger network to achieve convergence of the policy

The nonlinear activation function allows the network to be a *universal approximator*. It is this nonlinearity that allows this behavior as having a simple linear function of the input would result in the overall network being a linear function, as linear combinations of linear functions would result in a linear function. There are several kinds of activation functions, but in this work two in particular have been tested: the first is the *hyperbolic tangent* shown in Eq.2.29 and depicted in Fig. 2.3a , a continuous function with the advantageous property of having mean close to 0, while the second is the *Rectified Linear Unit* shown in Eq.2.30 and depicted in Fig. 2.3b, a piece wise continuous function. The former has been developed to improve convergence of the network, as it solves the exploding gradient problem in back propagation, thanks to having derivative with value either 1 or 0 (with other activation functions the chained multiplication of the gradients of each layer results in the parameters of the first layers updating extremely slow).

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.29}$$

$$ReLU(x) = max(0, x) \tag{2.30}$$



(a) Hyperbolic tangent ($tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$)

(b) ReLU - $ReLU(x) = max(0, x)$

Figure 2.3: Activation functions tested in the 3DOF scenario to select the one with the best performance and stability.

Chapter 3

Dynamics of the problem

In this chapter, the dynamics of the system is presented, both for the 3DOF and 6DOF cases, and a formulation of the landing problem suited to reinforcement learning is developed.

Assumptions

The rocket is modeled as a rigid body, with aerodynamic effects and a uniform gravitational field (a reasonable approximation due to proximity to Earth's surface).

The following assumptions are made:

- Center of mass (CoM) having a constant position in the body.
- Uniform and constant gravitational field.
- Negligible earth curvature and rotation effects.
- Fixed position of the center of pressure (CoP).
- Negligible change in thrust due to atmospheric pressure variation (from $2km$ to sea level the difference would only be about 2% if the thruster were to fire at maximum thrust).
- ISA atmospheric model Ref.[36] considered for atmospheric density.
- No sloshing in the propellant tanks.
- Pure drag aerodynamic force (no lift).
- No delays in the actuator's response.
- Diagonal inertia matrix.

3.1 3DOF dynamics

The system is first studied on a *simplified* 3DOF model of the rocket shown in Fig.3.1, modeling a planar trajectory. The dynamics equations are reported in equations 3.1, assuming also that:

- Only the axial force is considered (no *normal force*);
- Inertia computed using an average value for the mass;
- Gravity is uniform.

$$m\ddot{y} = T sin(\theta + \delta) - A \sin\theta - mg \tag{3.1}$$
$$m\ddot{x} = T cos(\theta + \delta) - A \cos\theta \tag{3.2}$$
$$I\dot{\omega} = -T \sin\delta \, (x_T - x_{CoM}) \tag{3.3}$$
$$\dot{m} = -\frac{T}{g_0 I_s} \tag{3.4}$$

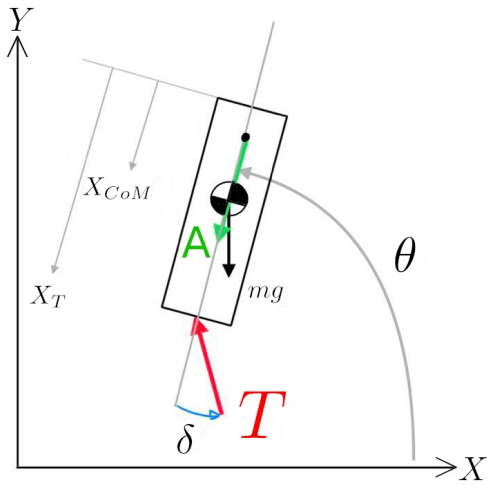


Figure 3.1: Reference frame used for the 3DOF dynamics and diagram of the forces acting on the launcher: weight mg (applied in X_{CoM}), thrust T (gimbaled by an angle δ and applied in X_T) and the aerodynamic axial force A (applied at the center of pressure). The Y axis points upwards.

Equations 3.1 are all expressed in the inertial reference frame. It is important to highlight that the 3DOF dynamics are expressed in a reference frame where

the y axis is the vertical one (i.e. the one expressing the altitude of the rocket) rather than the x axis as in the 6DOF dynamics.

The equations describing the dynamics of the system are assembled in a state space formulation describing the dynamics of the time-invariant system as in Eq.3.5

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \tag{3.5}$$

with the state vector $\boldsymbol{x}$ and the control vector $\boldsymbol{u}$ being:

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ \theta \\ v_x \\ v_y \\ m \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} \boldsymbol{\delta} \\ \boldsymbol{T} \end{bmatrix} \tag{3.6}$$

3.2 6DOF dynamics

In the second part of the thesis the 6DOF dynamics and environment are developed. This environment allows to train the agent in a realistic scenario, with both translational and rotational dynamics modeled.

3.2.1 Reference frames

Two reference frames are employed to define the equations of motion and the dynamics of the system, as depicted in Fig.3.2:

- **Inertial reference frame** $\mathcal{F}_I = [x_I, y_I, z_I]$: it is the non-accelerating frame fixed with the world, with the origin in the landing site and the x-axis pointing upward, the z-axis pointing north and y-axis pointing eastward.
- **Body-fixed reference frame** $\mathcal{F}_B = [x_B, y_B, z_B]$: this reference frame is fixed with the vehicle's body, centered at its center of mass (CoM) and it is aligned with its x-axis along the longitudinal axis of the vehicle, pointing towards the opposite end with respect to the thruster (i.e. towards the tip of the launch vehicle). The other two axes are perpendicular with respect to the x-axis but do not actually have a specific direction as the body of the rocket is assumed cylindrical.

If a generic vector $\boldsymbol{u}$ is expressed in a certain reference frame a it is denoted as $\boldsymbol{u}_a$ while a generic rotation matrix R from the reference frame a to the reference frame b is denoted as $R_{a\to b}$ (so converting a vector $\boldsymbol{x}_a$ in frame a to the same vector in frame b would be $\boldsymbol{x}_b = R_{a\to b}\boldsymbol{x}_a$).

Due to the choice of the reference systems in order to achieve a successful landing the condition to be met is to simply overlap the x-axis of reference frame $\mathcal{F}_B$ with $\mathcal{F}_I$.

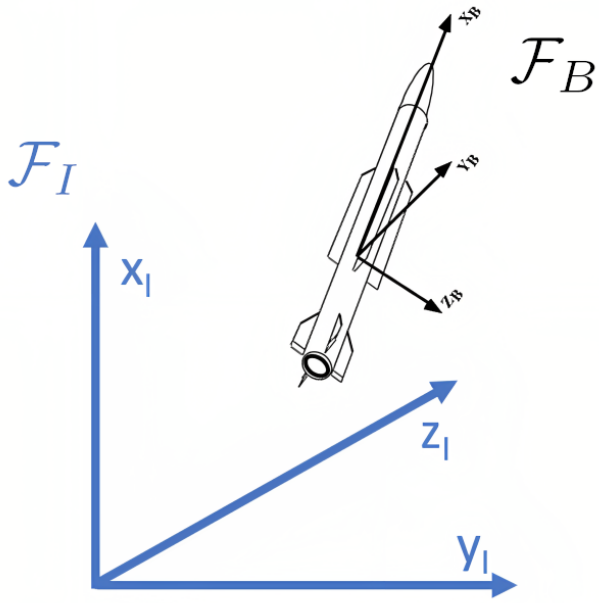


Figure 3.2: Depiction of inertial reference frame $\mathcal{F}_I$ (in blue) and body fixed reference frame $\mathcal{F}_B$ (in black). The x_I axis points upwards.

3.2.2 Translational dynamics

The translational dynamics are defined in the inertial reference frame $\mathcal{F}_I$ by the following system of differential equations:

$$\dot{\boldsymbol{r}}_I = \boldsymbol{v}_I \tag{3.7}$$

$$\dot{\boldsymbol{v}}_I = \frac{1}{m(t)}\boldsymbol{F}_I + \boldsymbol{g}_I \tag{3.8}$$

$$\dot{m} = -\frac{||\boldsymbol{T_I}||}{I_{sp}g_0} \tag{3.9}$$

where:

- $\boldsymbol{r}_I$ is the position vector of the center of mass in the inertial reference frame;
- $\boldsymbol{v}_I$ is the velocity vector of the center of mass;
- $m(t)$ is the time-varying mass of the vehicle (due to propellant consumption);
- I_{sp} is the specific impulse of the engine;
- g_0 is the standard gravitational constant at sea level;

- $\boldsymbol{g}_I = [-g_0\, 0\, 0]$ is the gravitational acceleration vector;
- $\boldsymbol{F}_I$ is the summation of the forces acting on the vehicle expressed in the inertial reference frame, detailed in 3.2.4.

In the following, the subscript I for the position and velocity vectors is dropped to simplify notation.

3.2.3 Rotational dynamics

The rotational dynamics are computed through Euler's rigid body equation and the kinematics are parametrized using the quaternion representation in the scalar-first convention:

$$\boldsymbol{q} = \begin{bmatrix} q_s \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} cos\frac{\theta}{2} \\ \boldsymbol{i} sin\frac{\theta}{2} \\ \boldsymbol{j} sin\frac{\theta}{2} \\ \boldsymbol{k} sin\frac{\theta}{2} \end{bmatrix} \tag{3.10}$$

In Eq.3.2.3 the angle θ represents *double* the rotation around the rotation axis expressed by the vector part of the quaternion $[q_x, q_y, q_z]$.

The rotation matrix from the inertial to the body reference frame can be computed from the elements of the quaternion as:

$$R_{I\to B} = \begin{bmatrix} 1-2(q_2^2+q_3^2) & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & 1-2(q_1^2+q_3^2) & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & 1-2(q_1^2+q_2^2) \end{bmatrix} \tag{3.11}$$

The dynamics is expressed in the *body-fixed reference frame* and the quaternion parametrization thus represents the attitude of the body-fixed frame $\mathcal{F}_B$ with respect to the inertial reference frame $\mathcal{F}_I$. The kinematics and dynamics equations are respectively Eq.3.12 and Eq.3.13.

$$\dot{\boldsymbol{q}} = \frac{1}{2}\boldsymbol{\Omega q} \tag{3.12}$$

$$\dot{\boldsymbol{\omega}}_B = \boldsymbol{J}^{-1}(\boldsymbol{M}_B - \boldsymbol{\omega}_B \times \boldsymbol{J}\boldsymbol{\omega}_B) \tag{3.13}$$

with the terms being:

- $\boldsymbol{\omega}_B = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^T$ the angular velocity vector expressed in body coordinates;

- $\boldsymbol{M}_B$ the summation of the moments acting on the rocket expressed in the *body frame*;
- $\boldsymbol{J}$ the inertia matrix of the rocket, computed at the beginning of each episode depending on the initial mass m of the system (which is an initial condition with a certain variability), the length of the rocket l and its base radius r:

$$\boldsymbol{J} = \begin{bmatrix} \frac{1}{2}mr^2 & 0 & 0 \\ 0 & \frac{1}{12}m(l^2+3r^2) & 0 \\ 0 & 0 & \frac{1}{12}m(l^2+3r^2) \end{bmatrix} \tag{3.14}$$

- $\boldsymbol{\Omega}$ the skew-symmetric matrix mapping the angular velocity and the current attitude quaternion to the derivative of the attitude quaternion.

$$\boldsymbol{\Omega} = \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{3.15}$$

3.2.4 Forces and moments

The two forces considered to be acting on the launcher are the control force $\boldsymbol{T}_I$ and the aerodynamic force $\boldsymbol{A}_I$, with the subscript I indicating that they're computed in the *inertial reference frame* $\mathcal{F}_I$. The two forces are actually first computed in the more natural *body reference frame* $\mathcal{F}_B$ and then expressed in the inertial one through a rotation:

$$\boldsymbol{F}_I = R_{B\to I}\left(\boldsymbol{T}_B + \boldsymbol{A}_B\right) \tag{3.16}$$

Control force

The rocket engine provides the necessary control force, gimbaling about the z_B body axis of an angle δ_Y (in the negative direction) and about the y_B body axis of an angle δ_Z (in the positive direction). The rotation matrix $R_{T\to B}$ rotates the thrust vector $\boldsymbol{T}_T = [T, 0, 0]^T$ from the thrust frame (i.e. the frame fixed with the nozzle axis) to the body frame:

$$R_{T\to B} = \begin{bmatrix} cos(\delta_y)cos(\delta_z) & -sin(\delta_y) & -cos(\delta_y)sin(\delta_z) \\ sin(\delta_y)cos(\delta_z) & cos(\delta_y) & -sin(\delta_y)sin(\delta_z) \\ sin(\delta_z) & 0 & cos(\delta_z) \end{bmatrix} \tag{3.17}$$

such that the thrust vector in the body frame $\mathcal{F}_B$ can be computed as:

$$\boldsymbol{T_B} = R_{T\to B}\boldsymbol{T_T} \tag{3.18}$$

The engine is modeled after the Merlin 1D of the Falcon 9 [37], it has saturations both in the gimbaling range being restricted to $\delta_z, \delta_y \in [-20°, +20°]$ and in the thrust magnitude having an upper bound at the maximum thrust achievable by the engine $T \in [0, 9.61kN]$.

Aerodynamic force

The simulator includes a model of the aerodynamic force, computed as:

$$\boldsymbol{A}_B = -\frac{1}{2}\rho||\boldsymbol{v}_I||S_A C_A R_{I\to B}\boldsymbol{v}_I \tag{3.19}$$

with $\rho = \rho(x)$ being the atmospheric density, $\boldsymbol{v}_I$ the velocity in the inertial reference frame $\mathcal{F}_I$, S_A the reference surface (in this work the base area of the rocket is used), C_A the aerodynamic coefficients matrix and $R_{I\to B}$ the rotation matrix from the inertial reference frame to the body frame $\mathcal{F}_I \to \mathcal{F}_B$.

The aerodynamic coefficients' matrix is a diagonal matrix with coefficients for each body axis:

$$C_A = \begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & c_z \end{bmatrix} \tag{3.20}$$

The aerodynamic coefficients are chosen to be equal, reflecting the so-called *spherical aerodynamic model*, in which the only aerodynamic force is the drag force. If the model had different aerodynamic coefficients for each axis then the aerodynamic force would not be exclusively antiparallel to the velocity vector, but there would also be the generation of a lift component (*elliptical aerodynamic model*) [8].

The atmosphere is modeled as having decaying density, which is computed according to the International Standard Atmosphere [36], considering the layer to be non-isothermal (as is the case in the altitude range selected):

$$\rho = \rho_b \left[\frac{T_b}{T_b + (h - h_b)L_b}\right]^{\left(1+\frac{g_0 M}{R^* L_b}\right)} \tag{3.21}$$

with the sea level density $\rho_0 = 1.225 kg/m^3$, the standard temperature $T_b = 288.15K$, the layer base height $h_b = 0m$, the *lapse rate* $L_b = -0.0065K/m$, the universal gas constant $R^* = 8.314\frac{Nm}{mol\,K}$, the standard gravitational acceleration $g_0 = 9.8067 m/s^2$ and the molar mass $M = 0.02897$kg/mol. The height h is actually the geopotential height but as the assumption is of constant gravity this can be considered as the actual geometric height.

State space formulation

As in the 3DOF case, the system is put in the nonlinear state space formulation with the state $\boldsymbol{x}$ and the control vector $\boldsymbol{u}$ being:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{r} \\ \boldsymbol{v} \\ \boldsymbol{q} \\ \boldsymbol{\omega} \\ m \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} \boldsymbol{\delta}_y \\ \boldsymbol{\delta}_y \\ \boldsymbol{T} \end{bmatrix} \tag{3.22}$$

The differential equations governing the dynamics of the system are thus represented as

$$\dot{\boldsymbol{x}} = \boldsymbol{f}\left(\boldsymbol{x}, \boldsymbol{u}\right)$$

3.3 Formulation of the problem

The planetary landing problem consists of reaching, starting from a certain set of initial conditions $\boldsymbol{x}_0$ of the system, a final state $\boldsymbol{x}_f$ with prescribed position and attitude and null terminal translational and rotational velocity. This should be achieved minimizing propellant consumption, in order to avoid carrying excessive propellant on the ascent phase of the mission (or the necessity of in-orbit refueling). In mathematical terms, the optimization problem in the 6DOF case can be expressed with the following system:

$$\begin{aligned}
&\underset{\boldsymbol{u}(t)}{\text{minimize}} && \Delta m = \int_{t_0}^{t_f} dm = m_0 - m_f && \text{(3.23a)}\\
&\text{subject to} && \dot{\boldsymbol{x}} = \boldsymbol{f}\left(\boldsymbol{x}, \boldsymbol{u}\right) && \text{(3.23b)}\\
&\text{with b.c.} && \boldsymbol{x}(t_0) = \boldsymbol{x}_0 && \text{(3.23c)}\\
& && \boldsymbol{r}(t_f) = \boldsymbol{r}_f && \text{(3.23d)}\\
& && \boldsymbol{v}(t_f) = \boldsymbol{v}_f && \text{(3.23e)}\\
& && \boldsymbol{q}(t_f) = \boldsymbol{q}_f && \text{(3.23f)}\\
& && \boldsymbol{\omega}(t_f) = \boldsymbol{\omega}_f && \text{(3.23g)}
\end{aligned}$$

with initial conditions sampled from a uniform distribution within a certain specified range $\boldsymbol{\rho}_{\boldsymbol{x}_0}$ (detailed for each run in the results section) and with the final conditions specified as:

$\boldsymbol{r}_f[m]$	$\boldsymbol{v}_f[m/s]$	$\boldsymbol{q}_f[-]$	$\boldsymbol{\omega}_f[rad/s]$
$\mathbf{0}$	$\mathbf{0}$	$[1, 0, 0, 0]$	$\mathbf{0}$

For the 3DOF case the formulation is the same, with the usage of the respective dynamics equations and the reduction in dimensionality of the vectors, moving from a quaternion parametrization of the attitude to just the angle θ.

$$\underset{\boldsymbol{u}(t)}{\text{minimize}} \quad \Delta m = \int_{t_0}^{t_f} dm = m_0 - m_f \tag{3.24a}$$

$$\text{subject to} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}_{\text{3DOF}}(\boldsymbol{x}, \boldsymbol{u}) \tag{3.24b}$$

$$\text{with b.c.} \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{3.24c}$$

$$\boldsymbol{x}(t_f) = \boldsymbol{x}_f \tag{3.24d}$$

with the final state:

$\boldsymbol{r}_f[m]$	$\theta_f[rad]$	$\boldsymbol{v}_f[m/s]$	$\omega_f[rad/s]$
$\mathbf{0}$	0	$\mathbf{0}$	0

In a reinforcement learning setting the constraints of the problem are not enforced in a strict way, but rather are hinted more or less strongly with the reward function, which is detailed in Section 4.5. This means that the landing problem can be expressed with a formulation employing soft constraints as:

- Minimize the used propellant mass Δm, the final position error $||\boldsymbol{r}_f||$, the final translational and rotational velocity error ($||\boldsymbol{v}_f||$ and $||\boldsymbol{\omega}_f||$ respectively) and the final attitude error $[\psi, \theta, \phi]_f$ (the attitude error is expressed in Euler angles as detailed in 4.5);
- With the lander subject to the *dynamics equation* $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$;
- With specified initial conditions $\boldsymbol{x}_0$.

This formulation is used as the foundation to develop the appropriate reward function which enforces the constraints by strongly penalizing their violation.

Chapter 4

Environment definition

In reinforcement learning the environment is a *Markov Decision Process* - MDP (or sometimes a *Partially Observable Markov Decision Process* - POMDP, if only an observation of the state is accessible), consisting of a set of possible states $\mathcal{S}$, a set of possible actions $\mathcal{A}$, a set of rewards $\mathcal{R}$, a transition function $\boldsymbol{s'} = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{a})$ mapping the next state $\boldsymbol{s'}$ to both the previous state $\boldsymbol{s}$ and the action taken $\boldsymbol{a}$ and a reward function assigning a scalar reward at each transition $R(\boldsymbol{s'}, \boldsymbol{s}, \boldsymbol{a})$.

4.1 Environment structure

The OpenAI Gym environment specification is employed to have a standardized environment in which the application to the rocket landing problem is tested. This offers the advantage of speeding up the development project as the interfaces are already defined. The core of the implementation is the environment defining both the state transition function (the dynamics of the rocket integrated through time) and the reward function.

The environment is built to be compliant with OpenAI Gym APIs [38], providing the following *standardized methods*:

- `.step()`: simulates the system dynamics for one time step and provides as output an observation, a reward (float), a `done` flag (boolean, 1 if the environment has reached a terminal state) and an info dictionary (utility variable to output useful information);
- `.reset()`: initializes the environment and returns an initial observation;
- `.render()`: output method that is used to provide to the user a visualization of the current condition of the environment (i.e. 3d graphic, plots,…);
- `.close()`: terminates and cleans-up the environment.

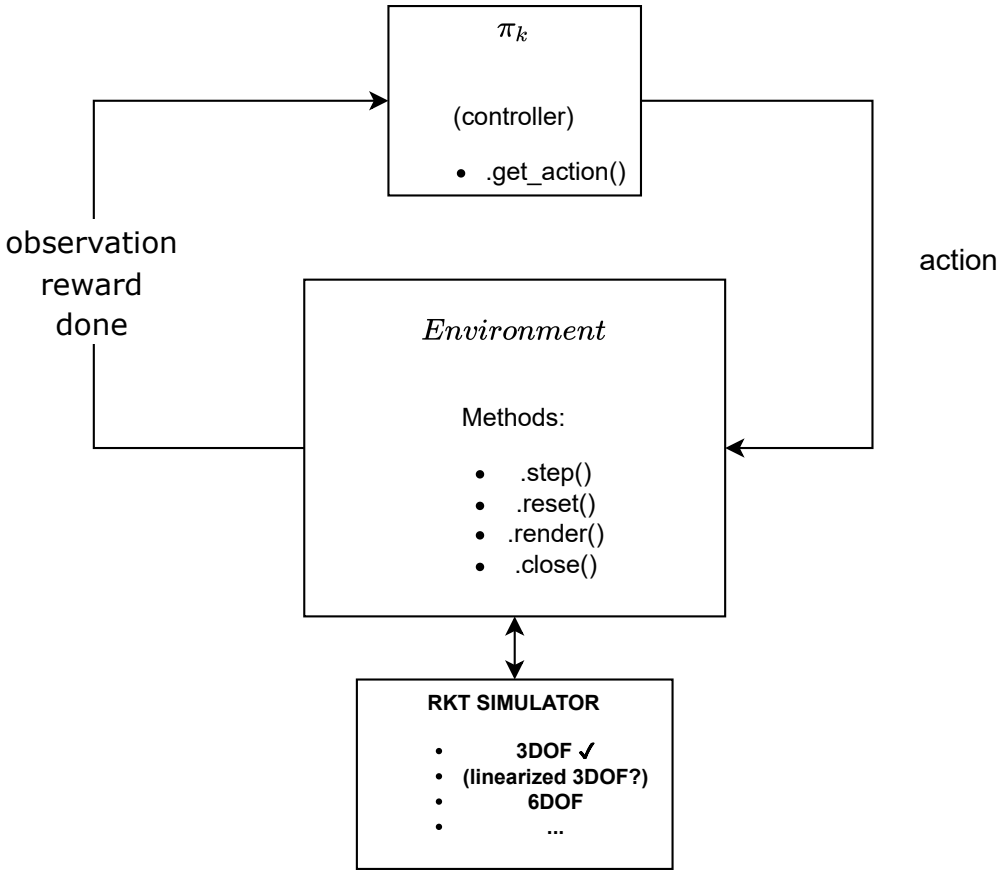


Figure 4.1: Schematic of the environment and its interfaces with the overall RL framework: the simulator can have different dynamics and communicates with the environment, which employs standard APIs to interface with the RL policies. This enables using standard RL frameworks for training.

A schematic of how the environment interacts with the policy can be seen in The dynamics equations are integrated using a variable-step RK45 ODE integrator from the Python package SciPy [39]. The relative and absolute tolerances (respectively ϵ_{rel} and ϵ_{abs}) are reported in Table 4.1

Type	ϵ_{rel}	ϵ_{abs}
Value	0.001	1e-06

Table 4.1: Absolute and relative tolerances of the ODE45 integrator

and the quaternion $\boldsymbol{q}$ is normalized at each integration step. The state transition function mapping the current state-action pair $(\boldsymbol{s}, \boldsymbol{a})$ to the next state $\boldsymbol{s}'$ can then be written as:

$$\boldsymbol{s}' = \boldsymbol{f}_{ODE45}(\boldsymbol{s}, \boldsymbol{a}) \tag{4.1}$$

The environment `.step()` method advances the simulation at a fixed timestep Δt while using the *event function* feature of SciPy's ODE45 integrator to check if zero height is reached; if so, the integration is stopped and the function outputs a zero height signal.

4.2 Observation Space

The observation space of the environment contains all possible values of its observations; similarly the state space is the set of all possible system states. In the case of the problem studied in this thesis, these are *continuous spaces* as the variables are real-valued. This type of space is supported by default by Gym, in the form of `Box` spaces. To improve convergence of the RL algorithms it is good practice to bound, if possible, the observation space to lie within a certain range and to normalize it.

Observation space normalization and bounding

In the defined environment some variables have natural bounds while others are theoretically unbounded. The state space (and thus the observation space) is normalized so that the bounds for each of its elements are $[-1, +1]$. Some reasonable bounds have been selected for each, while allowing exploration within a reasonable range of states.

The normalization vector is selected considering reasonable maximum values for each variable:

- **Free fall time**: the free fall time is used to have a heuristic for the duration of the episode in order to compute the other maximum values and is simply computed as

$$t_{\text{free fall}} = \frac{-v_x(t=0) + \sqrt[2]{v_x(t=0)^2 + 2 \cdot 9.81 \cdot x(t=0)}}{g_0}$$

- **maximum velocity**: it is considered to be twice the terminal velocity of the rocket in an atmosphere-less environment at a time equal to $t_{\text{free fall}}$

$$v_{\max} = 2 \cdot 9.81 \cdot t_{\text{free fall}}$$

  as the rocket would have a significant part ($\sim 100 \div 50\%$) of its velocity directed in the downward component at the beginning of each episode (due to the nature of the landing problem) and would not accelerate further in this direction using its thrusters (if this were to be the case it would be better to terminate the episode prematurely as it would mean the rocket is upside down and thrusting downwards, an unfeasible scenario)

- **maximum angular velocity**: is considered as the maximum angular velocity that the rocket would achieve if it were to thrust for the free fall time at the maximum gimbal angle and maximum thrust of the engine:

$$\omega_{\max} = \frac{T_{\max} \sin(\delta_y^{max}) ||\boldsymbol{r}_B^{\text{thrust}}||}{5 J_{xx}} t_{\text{free fall}}$$

with J_{xx} being the minimum inertia moment (around the x_B body axis) and $\boldsymbol{r}_B^{\text{thrust}}$ being the vector in body coordinates between the center of mass of the rocket and the engine gimbal point.

State	Normalizer
x	$1.5\|x_0\|$
y	$1.5\|y_0\|$
θ	2π
v_x	$v_{\max}$
v_z	$v_{\max}$
ω	$2\omega_{max}$
m	$m_0 + \Delta_m$

Table 4.2: Normalizer vector $\boldsymbol{y}_{\text{normalizator}}$ for the state in the 3DOF problem

3DOF state space normalization

In the 3DOF Gym environment the state returned by the simulator is normalized at each step by dividing element-wise the state by the maximum of the absolute value of the state bounds, i.e. a normalization vector as shown in Tab.4.2. Thus, the normalized state vector can be computed as Eq. 4.2.

$$\boldsymbol{y} \leftarrow \boldsymbol{y} \oslash \boldsymbol{y}_{\text{normalizator}} \tag{4.2}$$

The Observation Space is enforced by terminating the episode if any of the values of the state exceeds the state bounds.

6DOF state space normalization

In the 6DOF environment the *normalizer vector* $\boldsymbol{x}_{\text{normalizer}}$ for the 6DOF environment is thus defined in Table 4.3, by using the previous maximum values, considering that each quaternion component has at most value of 1 and that the rocket has maximum mass at the beginning of each episode ($m_0 + \Delta_{m_0}$ with Δ_{m_0} being the range of variation of m_0), before firing its engine.

Considering that some of its elements could be zero, the actual *normalizer vector* is updated by taking the element-wise maximum between the vector itself and 1, as in Eq.4.3.

$\boldsymbol{r}$	$x : 1.2 * \lvert x(t=0)\rvert \; y, z : 1.5\lvert y(t=0)\rvert, 1.5\lvert z(t=0)\rvert$
$\boldsymbol{v}$	$[v_{max}, v_{max}, v_{max}]$
$\boldsymbol{q}$	$[1, 1, 1, 1]$
$\boldsymbol{\omega}$	$[\omega_{max}, \omega_{max}, \omega_{max}]$
m	$m_0 + \Delta_m$

Table 4.3: Normalizer vector for 6DOF environment

$$\boldsymbol{x}_{\text{normalizer}} \leftarrow \max(\boldsymbol{x}_{\text{normalizer}}, 1) \tag{4.3}$$

Then, the state vector is normalized by dividing element wise by the *normalizer vector* as Eq.4.4.

$$\boldsymbol{x} \leftarrow \boldsymbol{x} \oslash \boldsymbol{x}_{\text{normalizer}} \tag{4.4}$$

4.2.1 Episode termination conditions

Each episode is run until either a time limit is reached or one of several *early termination conditions* are reached. The early termination due to the time limit is enforced by using the `TimeLimit` wrapper of the Gym library, which outputs a done signal if the number of steps is greater than a set limit. Furthermore, the episode is prematurely terminated if one of the following conditions is met:

- Zero height is reached during the integration;
- The upper or lower *position* bounds are reached, which for the 6DOF are selected to be $\pm 0.9 \max(\boldsymbol{r}_{\text{normalizer}}, 200)$ except for the lower position bound on the x axis (vertical height), which is set as $x_{\min} = -30m$, as the episode would be terminated anyway when the rocket reaches zero height by the event function. The *position* bound limit $\boldsymbol{r}_{\text{normalizer}}$ is taken from the observation normalized defined in Table 4.3.

4.3 Initial conditions

Two training sets of initial conditions have been used to test the algorithm: first a set of simplified initial conditions (lower height, lower velocity, velocity directed only downwards) and then a second set of initial conditions sourced from historic flight data of the Falcon 9, collected by *flightclub.io* [40]. This second set of data simulates landing on a downrange location, such as a barge in the middle of the ocean or a downrange landing pad.

Simplified initial conditions

For the simplified initial conditions the mean and range are reported in 4.4 in the case of the 3DOF environment and in 4.5 for the 6DOF case.

	$\boldsymbol{r}$	θ	$\boldsymbol{v}$	ω	m
$\boldsymbol{\mu}_{\boldsymbol{x}_0}$	$[50, 500]$ m	π rad	$[0, -50]m/s$	0 rad/s	$41e3$ kg
$\boldsymbol{\Delta}_{\boldsymbol{x}_0}$	$[5, 50]$ m	0 rad	$[0, 0]m/s$	0 rad/s	1000 kg

Table 4.4: Mean and range of simplified initial conditions for each state

	$\boldsymbol{r}\,[m]$	$\boldsymbol{v}\,[m/s]$	$\boldsymbol{q}\,[-]$	$\boldsymbol{\omega}\,[rad/s]$	$m\,[kg]$
$\boldsymbol{\mu}_{\boldsymbol{x}_0}$	$[500, 100, 100]$	$[-50, 0, 0]$	$[1, 0, 0, 0]$	$[0, 0, 0]$	$41e3$
$\boldsymbol{\Delta}_{\boldsymbol{x}_0}$	$[50, 10, 10]$	$[10, 10, 10]$	$[0.1, 0.1, 0.1, 0.1]$	$[0.1, 0.1, 0.1]$	$1e3$

Table 4.5: Mean and range of simplified initial conditions

Realistic initial conditions

The realistic initial conditions are shown in 4.6 in the 3DOF case and in 4.7 for the 6DOF.

	$\boldsymbol{r}$	θ	$\boldsymbol{v}$	ω	m
$\boldsymbol{\mu}_{\boldsymbol{x}_0}$	$[-1600, 2000]$ m	$\frac{3}{4}\pi$	$[180, -90]m/s$	0	$41e3$ kg
$\boldsymbol{\Delta}_{\boldsymbol{x}_0}$	$[200, 10]$ m	0.1	$[30, 30]m/s$	0.05	1000 kg

Table 4.6: Mean and range of initial conditions

	$\boldsymbol{r}\,[m]$	$\boldsymbol{v}\,[m/s]$	$\boldsymbol{q}\,[-]$	$\boldsymbol{\omega}\,[rad/s]$	$m\,[kg]$
$\boldsymbol{\mu}_{\boldsymbol{x}_0}$	$[2000, -1600, 0]$	$[-90, 180, 0]$	$[0.866, 0, 0, -0.5]$	$[0, 0, 0]$	$41e3$
$\boldsymbol{\Delta}_{\boldsymbol{x}_0}$	$[10, 200, 0]$	$[30, 30, 0]$	$[0.1, 0.1, 0.1, 0.1]$	$[0.05, 0.05, 0.05]$	$1e3$

Table 4.7: Mean and range of realistic initial conditions

4.4 Action Space

The action space comprises the space of all possible actions the agent can select. The actions are sampled from the policy $\pi_{\boldsymbol{\theta}_k}$, which outputs a mean value μ and a standard deviation σ: the action is then sampled from a normal distribution such

Action vector element	$\|\|\boldsymbol{T}_{max}\|\|$	δ_z	δ_y
Selected bound	9.61 kN	20 deg	20 deg

Table 4.8: Normalization values for the action

as $a = \mathcal{N}(\mu, \sigma)$. In reinforcement learning implementations it is preferred to have each term of the action bound in the interval $[-1, +1]$ to facilitate convergence of the algorithm: this means that each time the policy is sampled it will return a value between these bounds for each element. In the case studied, it is simple to enforce this condition as the action vector comprises three terms:

- **thrust** $||\boldsymbol{T}||$, which is limited to the range $||\boldsymbol{T}|| \in [0, 9.61]$ kN
- **gimbal angles** δ_z and δ_y, which are limited in the range $[-20\,\text{deg}, +20\,\text{deg}]$

The action space then simply becomes:

$$\mathcal{A} = \left\{ \boldsymbol{a} \in \mathbb{R}^3 : a_i \in [-1, +1], i = 1, 2, 3 \right\} \tag{4.5}$$

The action is denormalized when used in the simulator by simply multiplying it by the bounds vector.

4.5 Reward functions

One of the most important elements in the reinforcement learning paradigm is the reward function $R(\boldsymbol{s'}, \boldsymbol{s}, \boldsymbol{a})$. This function maps a tuple of next state $\boldsymbol{s'}$, current state $\boldsymbol{s}$ and action taken $\boldsymbol{a}$ to a scalar value r. This scalar value is used to map the value of each state and consequently to make the policy π converge to the optimal one π^*. One of the biggest issues in reinforcement learning is choosing a reward function that is reasonably descriptive for the problem at hand, meaning that it is not too *sparse*, otherwise this would make it impossible to map correctly the value of each state. Indeed, if in the landing problem the reward is only given upon landing in a state within the correct bounds the algorithm fails to converge to a policy different from a random one. Several iterations of the reward function have been tested to reach a satisfactory result, based on the idea of giving the agent a hint of the correct behavior in order to reach the final goal state and getting a bonus when this goal is achieved, while trying to minimize fuel consumption. The reward functions where first developed and tested on the 3DOF environment, allowing to quickly iterate its shape thanks to the lower run times of the algorithm in this environment, and then it is fine-tuned on the 6DOF environment to reach a satisfactory behavior in terms of convergence and performance of the obtained policy. In the following sections both versions are detailed.

Reward function development progression

Different reward functions were developed and tested, starting from the ones developed in [26] and [14]. The first rewards the agent for following a target velocity aiming for the landing site and giving a final bonus in case of successful landing. The second reward function has been developed as a subsequent step in the training of the neural network to be performed after convergence of the first training run: the agent learns first to land correctly by using the target velocity reward function and then a second optimization is performed by swapping the previous reward function for an annealed reward function, that rewards the agent mainly for achieving a successful landing while minimizing fuel consumption, without giving hints on the translation motion of the vehicle; in this way the first training run allows the agent to learn how to successfully land and only once the value function learns of the existence of a final bonus the minimization of fuel consumption is performed. Finally, to achieve a successful landing in a more realistic case of initial conditions a reward function based on following the acceleration command resulting from the energy-optimal solution of a simplified optimal control problem is developed. A recap of the steps is shown in Fig.4.2

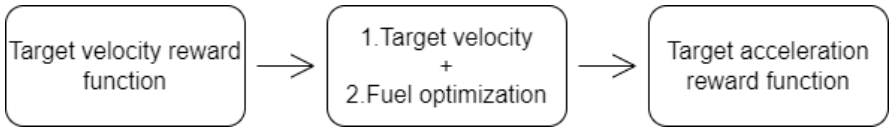


Figure 4.2: Steps in the development of reward functions: starting from targeting a *heuristic* velocity aiming for the landing site, we move to a two phases training to optimize more the fuel consumption. Finally, we move to a different reward function as the environment becomes more challenging

Attitude limits definition

In the 6DOF environment Euler angles are employed to define a negative reward for violating certain attitude bounds, in order to guide the agent towards a successful policy. The zyx extrinsic order of rotation convention is used, meaning that there is a first rotation around the inertial z axis of an angle ψ, then a rotation around the inertial y axis of an angle θ and finally a rotation around the x inertial axis of an angle ϕ, as shown in Fig.4.3. These bounds are defined by the θ_{lim} (limit angle) and θ_{mgn} (management angle) angles in the 3DOF case and by the $[\psi, \theta, \phi]_{\text{lim}}$ limit angles in the 6DOF environment. These are not strict constraints and their purpose is to avoid the agent exploring excessively states which are unlikely in the optimal solution.

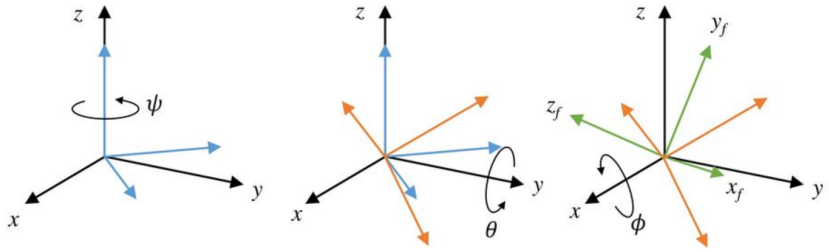


Figure 4.3: Euler angles rotation convention used in reward function to penalize exceeding certain attitude limits.

4.5.1 Target velocity reward

The first reward function that has been tested is the one from [26] and follows a Line-of-Sight heuristic to have at each time step a target velocity to hint both the direction in which the rocket should go and its speed. The reward then incentivizes strongly a successful landing by giving a final bonus when the landing conditions are satisfied and gives a penalty on thruster usage to minimize fuel consumption.

Both the 3DOF and 6DOF versions are shown in Eq.4.6 and Eq.4.7 respectively.

$$\begin{aligned} r =& \alpha||\boldsymbol{v} - \boldsymbol{v}_{targ}|| + \beta||\boldsymbol{T}_B|| + \gamma(|\theta| > \theta_{lim}) - \delta \cdot max(0, |\theta| - \theta_{mgn}) + \eta \\ &+ \kappa(z \leq 0 \, and \, ||\boldsymbol{r}_f|| < r_{lim} \, and \, ||\boldsymbol{v}_f|| < v_{lim} \, and \, |\omega| < \omega_{lim} \, and \, |\theta| < \theta_{lim_f}) \end{aligned} \tag{4.6}$$

$$\begin{aligned} r =& \alpha||\boldsymbol{v} - \boldsymbol{v}_{targ}|| + \beta||\boldsymbol{T}_B|| + \gamma \cdot any([|\psi|, |\theta|, |\phi|] > [\psi, \theta, \phi]_{lim}) + \eta \\ &+ \kappa(x \leq 0 \, \text{and} \, ||\boldsymbol{r}_f|| < r_{lim} \, \text{and} \, ||\boldsymbol{v}_f|| < v_{lim} \, \text{and} \, all(\boldsymbol{\omega} < \omega_{lim}) \\ &\text{and} \, all([\psi, \theta, \phi]_f < [\psi, \theta, \phi]_{lim_f})) \end{aligned} \tag{4.7}$$

Each term of the reward function is weighted and has a specific meaning:

- α: rewards having a velocity vector $\boldsymbol{v}$ close to the target velocity $\boldsymbol{v}_{targ}$;
- β: penalizes usage of the thrust, in order to reduce propellant consumption;
- γ: penalizes exceeding the limits on the Euler angles for attitude;
- η: this is a positive constant to avoid early termination by the agent (as all other rewards except for the terminal one are negative);
- δ: this term, present only in the 3DOF case (Eq.[4.6]), gives the agent a hint that it is approaching a limit in the attitude, defined by the θ_{mgn} angle;

Coefficient	α	β	γ	δ	κ	η	τ_1	τ_2
Value	-0.01	$-1e-7$	-10	-5	10	0.05	$20s$	$100s$

Table 4.9: Target velocity reward function coefficients

Coefficient	$\lvert\theta_{lim}\rvert$	θ_{mgn}	ω_{lim}	r_{lim}	v_{lim}	$[\psi, \theta, \phi]_{\text{lim}}$	$[\psi, \theta, \phi]_{\text{lim}_f}$
Value	$360°$	$180°$	$0.2rad/s$	$30m$	$10m/s$	$[85, 85, 360]°$	$[10, 10, 360]°$

Table 4.10: Target velocity reward function parameters

- κ: multiplies the *terminal reward term*, given to the agent only if the final landing conditions are satisfied.

For a more detailed overview of each term the reader is referred to [26].

The weights have been carefully tuned through subsequent runs of the algorithm starting from the values in [26], and the two scaling times τ_1 and τ_2 are taken from the reference implementation. Their values have been tweaked aiming to have each term with the same order of magnitude, except for the terminal bonus coefficient k, and are detailed in Table 4.9. The values of the attitude and final landing state parameters are shown in Table 4.10.

The target velocity magnitude follows a decaying exponential shape as shown in Eq.4.8 starting from the initial velocity $v_0 = ||\boldsymbol{v}(t_0)||$, based on the time-to-go computed in 4.9 dividing the distance to the landing pad by the speed at a given time. This target velocity targets a waypoint at $z^{\text{waypoint}} = 50m$ and is made to be only vertical when below this waypoint height thanks to the shape of $\hat{\boldsymbol{r}}$ as in Eq.4.10 and of $\hat{\boldsymbol{v}}$ as in Eq.4.11. The decay speed is controlled by the scaling time τ which also changes in value at the waypoint, as in 4.12

$$\boldsymbol{v}_{targ} = -v_0 \left(\frac{\boldsymbol{r}}{||\hat{\boldsymbol{r}}||} \right) \left(1 - exp\left(-\frac{t_{go}}{\tau} \right) \right) \tag{4.8}$$

$$t_{go} = \frac{\hat{\boldsymbol{r}}}{\hat{\boldsymbol{v}}} \tag{4.9}$$

$$\hat{\boldsymbol{r}} = \begin{cases} \boldsymbol{r} - [0\ 0\ z^{\text{waypoint}}], & \text{if } r_z > z^{\text{waypoint}} \\ [0\ 0\ r_z], & \text{otherwise} \end{cases} \tag{4.10}$$

$$\hat{\boldsymbol{v}} = \begin{cases} \boldsymbol{v} - [0\ 0\ 2], & \text{if } r_z > z^{\text{waypoint}} \\ \boldsymbol{v} - [0\ 0\ 1], & \text{otherwise} \end{cases} \tag{4.11}$$

$$\tau = \begin{cases} \tau_1, & \text{if } r_z > z^{\text{waypoint}} \\ \tau_2 & \text{otherwise} \end{cases} \tag{4.12}$$

Coefficient	ξ	γ	κ	δ	η
Value	0.004	-10	10	-5	0.05

Table 4.11: Annealed reward function coefficients

4.5.2 Annealed reward function

To improve the fuel efficiency of the solution a second training run is performed by retraining the model which converged using the target velocity reward function with a different reward function without the target velocity term, which penalizes fuel consumption and rewards successful landings. This allows the agent to first discover the final bonus in the first training run and then, once it achieves the goal conditions consistently the shaped part of the reward function is removed, to incentivize only the minimization of fuel consumption. The terms that penalize excessive flight path angles are kept to avoid instabilities in the training process. The function is shown in the 3DOF version in Eq.4.13 and in its 6DOF version in Eq.4.14. The coefficients for this reward function are shown in Table 4.11.

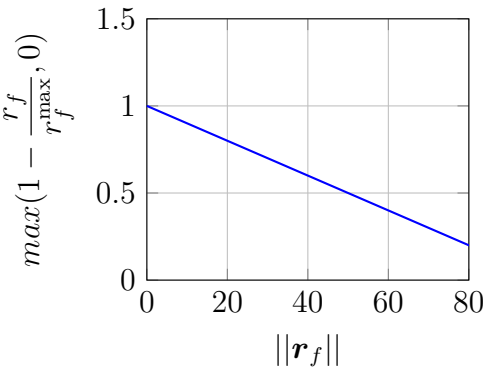
$$\begin{aligned} r = {} & \xi(||\boldsymbol{T}||) + \gamma(|\theta| > \theta_{lim}) - \delta \cdot max(0, |\theta| - \theta_{mgn}) + \eta \\ & + \kappa(z \leq 0 \, and \, ||\boldsymbol{r}|| < r_{lim} \, and \, ||\boldsymbol{v}|| < v_{lim} \, and \, |\omega| < \omega_{lim} \, and \, |\theta| < \theta_{lim_f}) \end{aligned} \tag{4.13}$$

$$\begin{aligned} r = {} & \xi(||\boldsymbol{T}||) + \gamma \cdot any([|\psi|, |\theta|, |\phi|] > [\psi, \theta, \phi]_{lim}) + \\ & + \kappa(x \leq 0 \, and \, ||\boldsymbol{r}|| < r_{lim} \, and \, ||\boldsymbol{v}|| < v_{lim} \, \text{and} \, all(\boldsymbol{\omega} < \omega_{lim}) \\ & \text{and} \, all([\psi, \theta, \phi]_f < [\psi, \theta, \phi]_{lim_f})) \end{aligned} \tag{4.14}$$

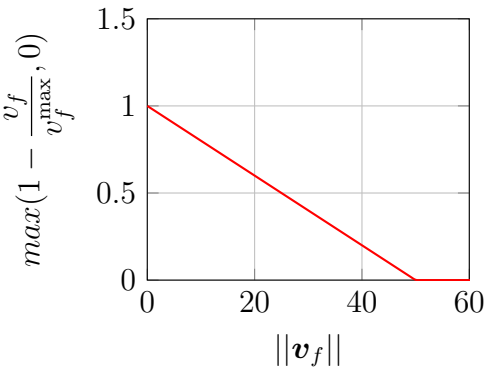
4.5.3 Target acceleration reward

The target velocity reward function turns out to work well in the simplified initial conditions case but does not reach a successful landing in the case of realistic initial conditions, with an excessive vertical component of the velocity vector and a landing tilt angle too high. A different reward based on a target acceleration has been developed to reach a correct landing with these initial conditions. The reward function gives a hint to the agent to follow a target acceleration $\boldsymbol{a_{targ}}$, which acts as a proxy to obtain a quasi-optimal policy and is detailed in the following paragraphs.

The reward function has been selected as Eq.4.15 for the 3DOF case and as Eq.4.16 for the 6DOF case:



(a) Terminal position reward



(b) Terminal velocity reward

Figure 4.4: Shape of the terminal reward bonuses: as the terminal position and velocity errors decrease the bonus increases up to 1, as the two functions are multiplied together, and is then multiplied by a scaling weight w_f.

Coefficient	α	β	γ	δ	η	w_f	$v_f^{\max}$	$v_f^{\max}$	κ
Value	-0.01	$-1e-8$	-1	-5	0.1	50	$50m/s$	$100m/s$	10

Table 4.12: Target acceleration reward function coefficients

$$
\begin{aligned}
r = {} & \alpha||\boldsymbol{a} - \boldsymbol{a}_{targ}||\Delta_h + \beta||\boldsymbol{T}|| + \eta \\
& + \gamma(|\theta| > \theta_{lim}) - \delta \cdot max(0, |\theta| - \theta_{mgn}) + \eta \\
& + w_f \cdot max\left(1 - \frac{r_f}{r_f^{\max}}, 0\right) \cdot max\left(1 - \frac{v_f}{v_f^{\max}}, 0\right) \\
& + \kappa(x \leq 0\, and\, ||\boldsymbol{r}|| < r_{lim}\, and\, ||\boldsymbol{v}|| < v_{lim}\, and\, and\, |\omega| < \omega_{lim}\, and\, |\theta| < \theta_{lim_f})
\end{aligned}
\tag{4.15}
$$

$$
\begin{aligned}
r = {} & \alpha||\boldsymbol{a} - \boldsymbol{a}_{targ}||\Delta_h + \beta||\boldsymbol{T}|| + \eta \\
& + \gamma any([|Yaw|\, |Pitch|\, |Roll|] > [Yaw\, Pitch\, Roll]_{lim}) \\
& + w_f \cdot max\left(1 - \frac{r_f}{r_f^{\max}}, 0\right) \cdot max\left(1 - \frac{v_f}{v_f^{\max}}, 0\right) \\
& + \kappa(x \leq 0\, and\, ||\boldsymbol{r}|| < r_{lim}\, and\, ||\boldsymbol{v}|| < v_{lim}\, all(\boldsymbol{\omega} < \omega_{lim}) \text{and}\, all(\boldsymbol{q} < \boldsymbol{q}_{\text{lim}}))
\end{aligned}
\tag{4.16}
$$

This approach tries again to achieve a balance between giving the agent a hint to achieve a successful landing and minimizing fuel consumption. The reward coefficients are detailed in Table 4.12.

In this case to improve convergence there are also two terms which give a linearly decaying reward as the final velocity and position errors increase within a certain interval. They are multiplied together and scaled by the weight w_f and their plots are shown in Fig.4.4. Other kind of functions have been tested to shape the terminal reward, such as quadratic and exponential functions but the linear ones resulted in the smallest landing errors. The target acceleration reward term is given down to a certain waypoint height and then set to 0 to let the agent explore more, in fact this can be modeled as being multiplied by a step function of the height Δ_h (in the 3DOF case the height variable is z rather than x):

$$\Delta_h = \begin{cases} 1 & \text{if } x >= waypoint \\ 0 & \text{if } x < waypoint \end{cases} \tag{4.17}$$

The waypoint is set to $x = 50m$ The target acceleration a_{targ} is the solution of the simplified problem minimizing the energy performance index in Eq 4.18.

$$J = \frac{1}{2}\int_{t_0}^{t_f} \boldsymbol{a}^T \boldsymbol{a}\, dt \tag{4.18}$$

subject to the following dynamics:

$$\dot{\boldsymbol{r}} = \boldsymbol{v} \tag{4.19}$$
$$\dot{\boldsymbol{v}} = \boldsymbol{g} + \boldsymbol{a} \tag{4.20}$$
$$\boldsymbol{a} = \frac{\boldsymbol{T}_I}{m} \tag{4.21}$$
$$\tag{4.22}$$

In the case studied, both terminal position and velocity are null, thus $\boldsymbol{r}_f = \mathbf{0}$ and $\boldsymbol{v}_f = \mathbf{0}$ resulting in the following boundary conditions:

$$\boldsymbol{r}(t_0) = \boldsymbol{r}_0 \;\; \boldsymbol{r}(t_f) = \boldsymbol{r}_f = \mathbf{0} \tag{4.23}$$
$$\boldsymbol{v}(t_0) = \boldsymbol{v}_0 \;\; \boldsymbol{v}(t_f) = \boldsymbol{v}_f = \mathbf{0} \tag{4.24}$$
$$\tag{4.25}$$

This problem has the analytical solution:

$$\mathbf{a}_{targ} = -\frac{6\mathbf{r}}{t_{go}^2} - \frac{4\mathbf{v}}{t_{go}} - \mathbf{g} \tag{4.26}$$

with the time-to-go t_{go} computed as the real positive solution of the quartic equation

$$g^2 t_{go}^4 - 4||\mathbf{v}||^2 t_{go}^2 - 24\mathbf{r}^T\mathbf{v}t_{go} - 36||\mathbf{r}||^2 = 0 \tag{4.27}$$

This problem does not account for the presence of the atmosphere and is not fuel-optimal, however it is used as a proxy to achieve a successful landing by computing

the optimal acceleration at each time step and using it as a target to balance the other terms of the reward function. To avoid an excessive acceleration command for the engine the actual target acceleration is computed as:

$$\boldsymbol{a}_{targ} = sat_{T_{max}/m}(\boldsymbol{a}_{targ}) \tag{4.28}$$

where the saturation function is:

$$sat_U(\boldsymbol{q}) = \begin{cases} \boldsymbol{q} & \text{if } ||\boldsymbol{q}|| \leq U \\ \boldsymbol{q}\frac{U}{\boldsymbol{q}} & \text{if } ||\boldsymbol{q}|| > U \end{cases} \tag{4.29}$$

A detailed solution to the problem can be found in [41]. This target acceleration has been exploited also in [14], albeit with a different action space and a different learning algorithm.

4.6 Software setup and execution pipeline

The PPO implementation used is the one from Stable Baselines3 [42], which has been validated against the relevant papers to ensure a robust implementation of the algorithm. This library also provides several useful tools to monitor the training runs such as a wrapper for the environment to log relevant data to cloud services and functions to check the functionality and adherence to the Gym standard of the developed environment.

The training runs were initially executed on local machines but when the computational burden became more relevant the training runs have been scaled up to be run in parallel on *virtual machines* (VMs) provisioned by Google Cloud. A pipeline to continuously check and run new experiments pushed to a specific branch of the GitHub repository was set up using GitHub Actions, a CI/CD (Continuous Integration/Continuous Deployment) framework.

Both the local machine (used for development and testing) and the Cloud VMs run the software in a Conda Python environment which isolates all the libraries and package dependencies to ensure reproducibility across machines and to increase reusability of the code.

In the end all the *artifacts*, i.e. the results from the training run (Neural Network model, trajectory plots, reward and state vector plots, etc.) are uploaded to Wandb (Weights and Biases [43]), a crucial tool used to visualize in any web browser the results of the training runs. This tool has several useful features, being able to create a dashboard of metrics and plots on the base of which it is possible to evaluate an experiment and compare several training runs. The execution pipeline for each training run is detailed in Fig.4.5.

Furthermore, the `.render()` method has a full visualization of the environment state, using in the 3DOF case the 2D visualization library Pygame and in the 6DOF [44], a full 6DOF OpenGL rendering engine with support for arbitrary

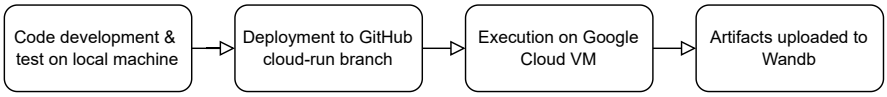
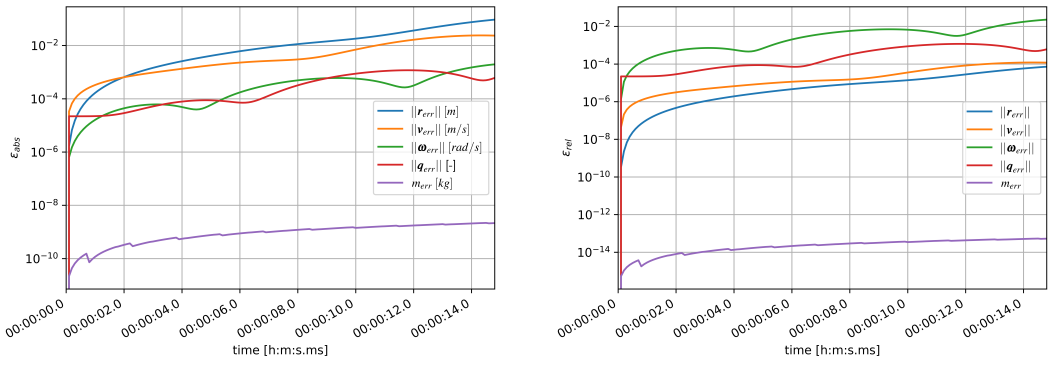


Figure 4.5: Training run continuous execution pipeline; this pipeline allows to quickly deploy an experiment to the Cloud VM and analyze the results of the training run when it is complete.

meshes, lighting and shadows effects and much more. These visualization tools revealed to be priceless when debugging the environment and designing the reward functions.

4.7 Environment validation

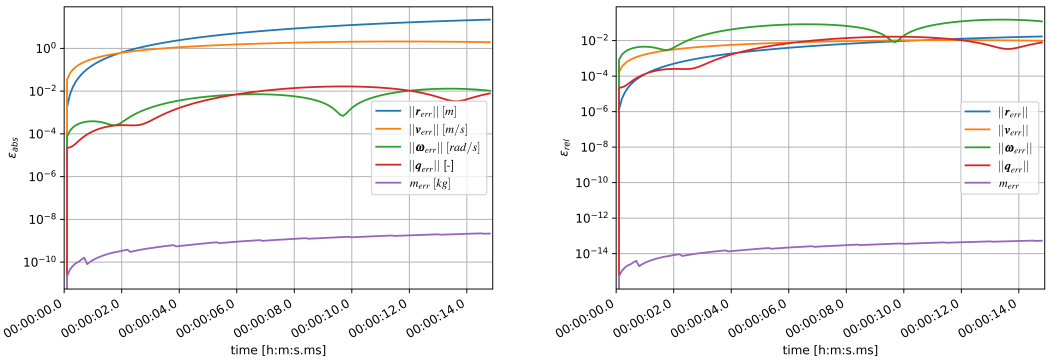
The dynamics of the environment and the integrator have been validated using as baseline the simulator developed in [9] which had in turn been validated using the FES flight environment simulator internal software from Deimos Space. The validation was performed simulating several trajectories with different control actions applied and checking the error between the two simulators. The samples shown hereafter input a constant actuator command (constant gimbal angles and thrust $\boldsymbol{u} = \begin{bmatrix} 5° & 5° & 98000N \end{bmatrix}^T$) and are shown to highlight a worst-case scenario in terms of trajectory, as extremely high rotational velocities are reached. Other trajectories were tested to validate the behavior on more realistic test cases (low angular velocities) and the errors are several orders of magnitude lower the (already small) ones reported here. Overall the simulator manages to capture faithfully the relevant behavior of the launcher.



(a) Absolute error (b) Relative error

Figure 4.6: The errors between the validated and the developed simulator are extremely low, showing that the dynamics are correctly represented.

To validate the simulator the same atmospheric model used in [9] is first employed in order to correctly assess the magnitude of the errors. Both the relative and absolute errors are plotted for all state variables in Fig.4.6. The relative and absolute errors are quite tiny and through a deeper analysis it's assessed that the difference stems mainly from differences in the normalization of the quaternion. This reflects in a *slight* discrepancy in the rotational behavior which is however emphasized in this validation run due to the nature of the trajectory employed: the launcher is accelerating to an extremely high angular velocity due to the constant gimbaled thrust, a behavior which would actually result in premature termination of the episode by the environment.



(a) Absolute error

(b) Relative error

Figure 4.7: Errors between validated and developed simulator with ISA atmosphere. In this case the errors are slightly higher due to the difference in atmospheric model, but still show that the developed simulator faithfully represents the dynamics of the problem.

The simulator employed for the training phase uses a more appropriate atmospheric model (ISA atmosphere), thus another validation has to be performed in order to assess the difference between the two models. As shown in Fig.4.7 the atmospheric model does not have a significant effect on the behavior of the launcher and the error from the validated simulator is still low. In particular from Fig.4.7b is evident that the relative errors are small, as expected due to the small difference in density difference ($\sim 3 \div 5\%$ at most, with a reduction in the difference as altitude decreases).

Chapter 5

Results 3DOF environment

In this chapter the results of the 3DOF case are shown, quantifying both the performance of the Reinforcement Learning algorithm and the performance of the obtained trajectory.

The planetary landing problem was first studied on a planar case, in which the dynamics considered are constrained on a plane and there are only 3 degrees of freedom: 2 translational and 1 rotational. Two training runs are shown: first for a set of simplified initial conditions (lower height, lower velocity, velocity directed only downwards) and then a second set of initial conditions sourced from historic flight data of the Falcon 9, collected by *flightclub.io* [40]. This second set of data simulates landing on a downrange location, such as a barge in the middle of the ocean or a downrange landing pad. Furthermore, initially in this simplified environment a discrete action space was tested, to test the algorithm's behavior on a simpler problem. This incremental approach was critical in reaching successful results in the more general, continuous case, in particular for iterating quickly on the shaped reward function and the hyperparameters.

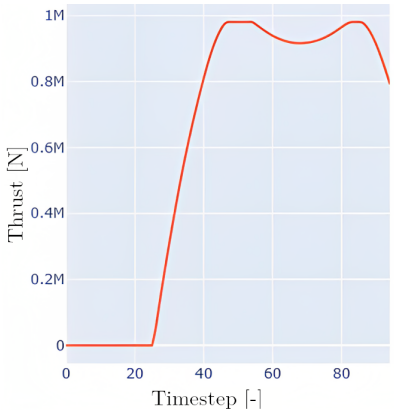
5.1 Simplified initial conditions

For the simplified initial conditions a lower initial height and initial velocity are selected. Furthermore, the velocity is *nominally* in the vertical direction only. The mean and range of the distribution are reported in Table 4.4.

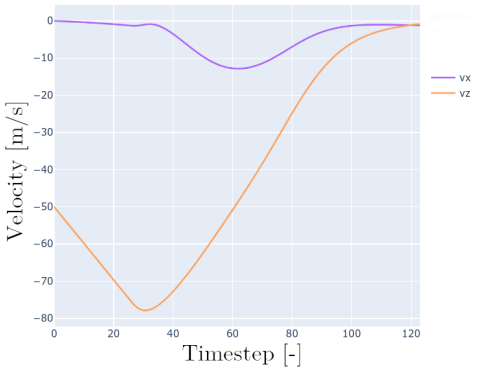
It is interesting to analyze the profiles of velocity, in Fig.5.1b, and thrust, in Fig.5.1a, of a sample episode after convergence.

The two figures two show that the agent tries to maximize the reward by using the thrusters at a minimum level, thus gaining speed, and then performing a high-thrust final burn. This reflects the ideal thrust profile, as the launcher avoids having high gravity losses during the landing burn by employing the atmospheric drag to do part of the work required along the trajectory.

In Fig.5.2 the convergence behavior of the algorithm is highlighted. Due to



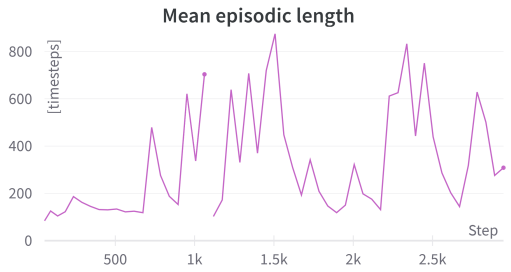
(a) Thrust profile of a sample episode



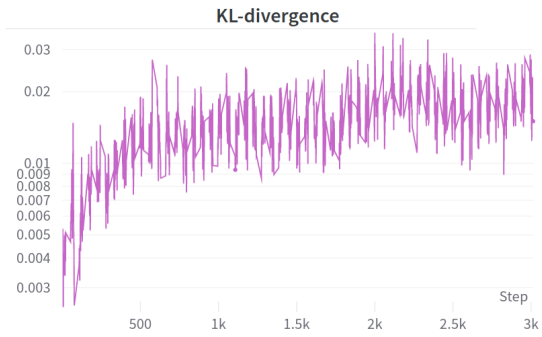
(b) Speed profile of a sample episode

Figure 5.1: Thrust and velocity profiles of the converged policy, showing the agent using a bang-bang profile to minimize gravity losses during the final burn.

(a) Mean episodic reward



(b) Mean episodic length (time step $\Delta t = 0.1s$)



(c) Approximate KL-divergence

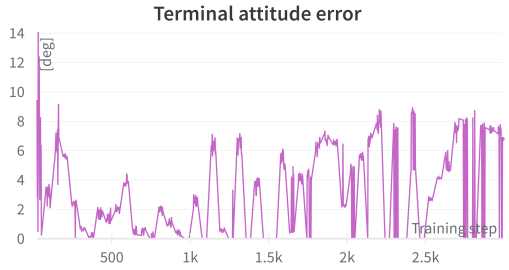
(d) Training loss (i.e. of the NN)

Figure 5.2: Trend of RL training metrics through training (simplified initial conditions), showing convergence to a policy maximizing mean reward and minimizing episodic length. The spikes show the exploration process of RL.

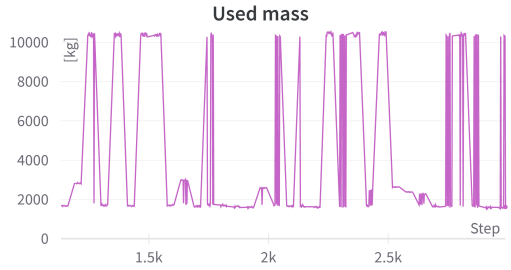
the multiphase training process there are discontinuities in the mean reward and episodic lengths at the 1k Step mark. The exploratory behavior of the algorithm

results in spikes due to deviation from the locally optimal policies, which are crucial to improve the performance of the policy and converge towards the *global* optimum.

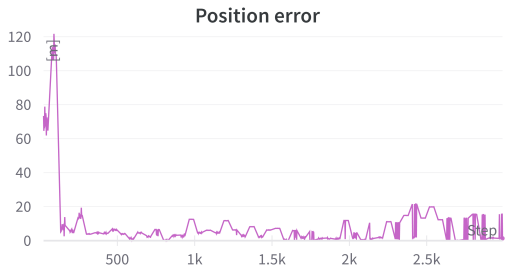
The convergence can also be analyzed by looking at the end-of-episode metrics (i.e. the terminal errors and used propellant mass). These are shown in Fig. 5.3 and show quick convergence towards low terminal errors in both position (Fig. 5.3c), velocity (Fig.5.3d) and attitude (Fig. 5.3a), with a gradual optimization of the used propellant mass as shown in Fig.5.3b.



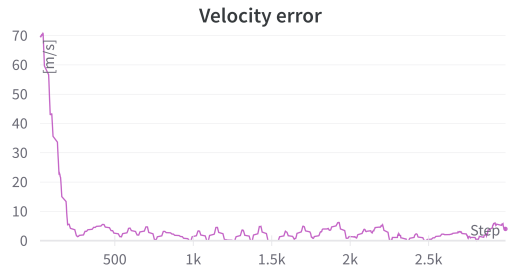
(a) Terminal attitude error



(b) Used propellant mass



(c) Position error



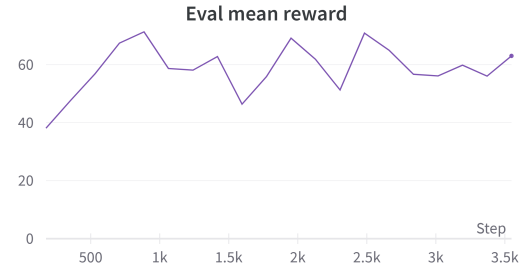
(d) Velocity error

Figure 5.3: Trajectory metrics during training (simplified initial conditions). The position and velocity error steeply go down and the used mass trends downwards when outliers are not considered.

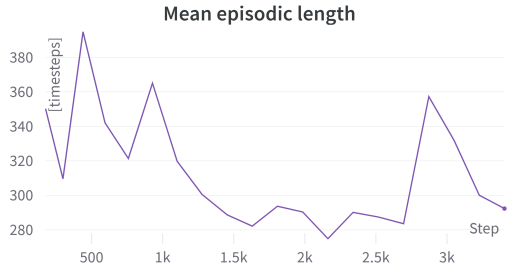
5.2 Realistic initial conditions

In this training run the initial conditions were sampled from a mean $\boldsymbol{\mu}_{\boldsymbol{x}_0}$ and a range $\boldsymbol{\Delta}_{\boldsymbol{x}_0}$ reported in Table 4.6.

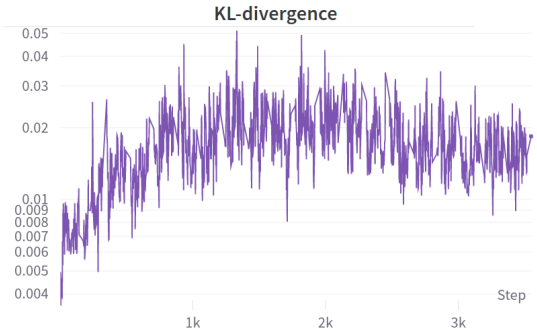
In Fig.5.4 the training metrics measuring the performance of the algorithm are shown. There is rapid convergence to a good value of mean reward, evaluated periodically from a batch of trajectories runs during training. By good values it is meant a reward value close to the theoretical maximum, which can be approximated by considering a case in which the target acceleration is tracked perfectly, and the thruster is not used (clearly this is not actually achievable, but it is useful to provide an upper bound), thus receiving only the terminal bonuses.



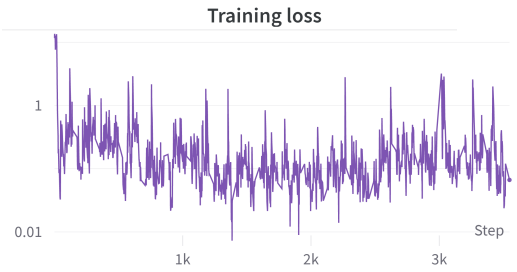
(a) Mean reward at evaluation



(b) mean episodic length



(c) Approximate KL-divergence

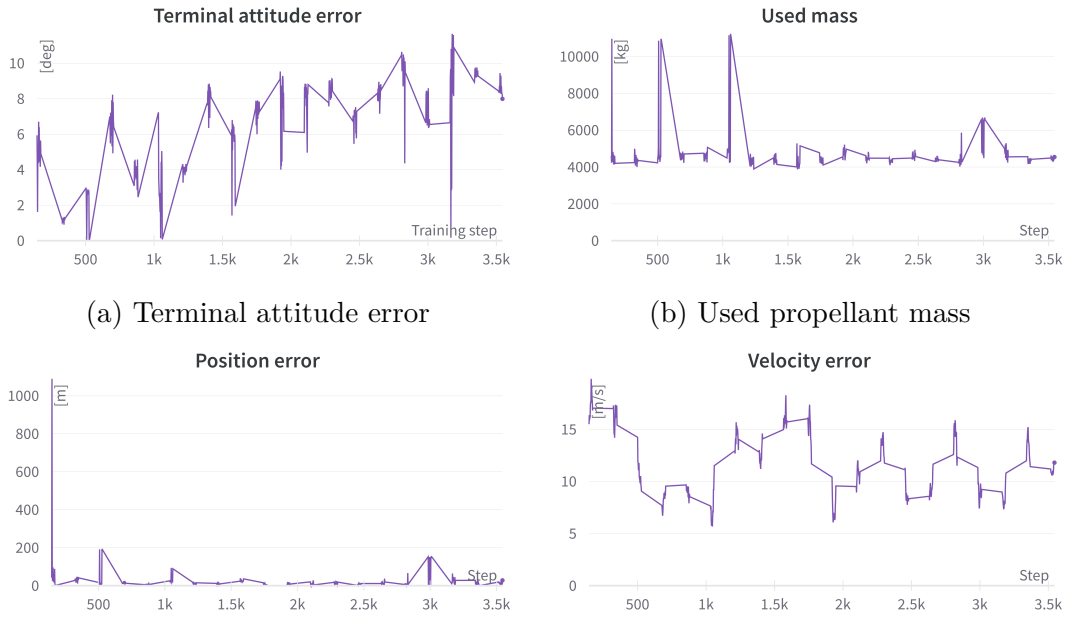


(d) Training loss (i.e. of the NN)

Figure 5.4: Trend of RL training metrics through training (realistic initial conditions). The mean reward trends upwards with episodic length going down to minimize propellant consumption.

While the previous metrics assess the performance of the algorithm it is also important to assess how this reflects on the quality of the trajectory that is obtained employing the trained policy. This can be assessed by looking at the terminal errors and the used mass. These two metrics, combined with the reward trend, helped in tweaking the reward function.

The algorithm manages to converge to a robust policy, with low terminal velocity Fig. 5.5d and position errors Fig. 5.5c. The attitude error at touchdown



(a) Terminal attitude error

(b) Used propellant mass

(c) Terminal position error

(d) Terminal velocity error

Figure 5.5: Trajectory metrics during training (realistic initial conditions). We can notice that the velocity error does not go below a mean minimum of about $5m/s$.

is also acceptable. Overall all parameters are within reasonable bounds to consider the landings successful, and the dispersion is reasonably low. During training there are spikes in the errors due to the exploratory behavior, however the policy quickly converges again to low terminal errors.

5.3 Comparison of activation functions

Two types of activation functions have been tested with this environment: the Rectified Linear Unit ($ReLU$) and the Hyperbolic Tangent ($tanh$), both detailed in Section 2.3.4. Using ReLUs results in quicker convergence as seen in Fig.5.6, however there is a curious behavior of policy unlearning in which the reward peaks after a certain amount of training steps. This effect can be overcome by sampling the mean reward obtained by the policy and saving the best model, through periodic evaluation of the policy. Otherwise, the ReLU activation function requires limiting the KL-divergence, as not doing so would make the policy diverge. This aims to limit the change in policy parameters at each update, by using early stopping of the neural network optimizer. A good value for the limit has been empirically found to be $KL_{\text{targ}} = 0.01$, which does not slow down excessively the learning process but still allows it to reach the maximum episodic reward. These

tests are shown for the realistic initial conditions case in Fig.5.7. If monotonic behavior in training is desired, it can be (roughly) obtained by limiting the KL-divergence.

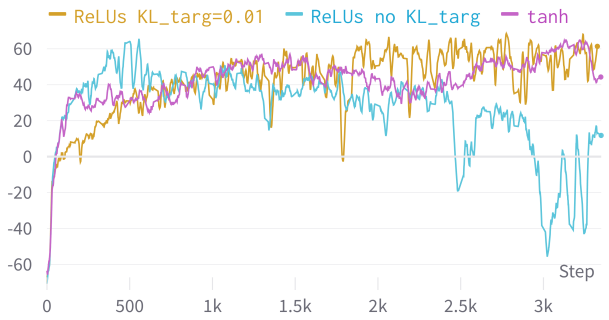
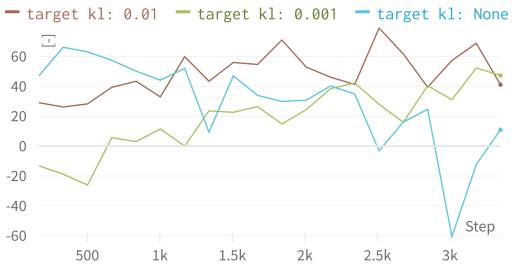
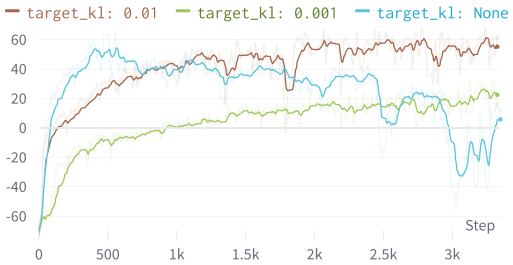


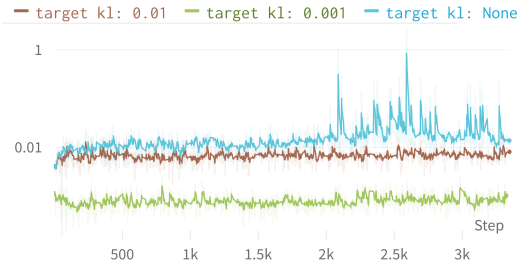
Figure 5.6: Comparison of different activation functions (mean reward through a rollout) showing the higher convergence speed of ReLU but also the need to limit the policy updates (through the KL-divergence target) to avoid policy-unlearning



(a) Mean reward at evaluation



(b) Mean reward through a rollout



(c) Approximate KL-divergence

Figure 5.7: Effects of constraining the KL-divergence for ReLUs activation functions: if left unconstrained the policy suffers an un-learning behavior after reaching a maximum in the reward.

Overall the advantages ReLUs have in regard to speed of convergence present also the downside of convergence instability. For the 6DOF environment some

tests were performed with this activation function, but it was found that the instability in convergence prevented successfully finding good policies. Thus, the decision was made to use the hyperbolic tangent activation function.

Chapter 6

Results 6DOF environment

In this chapter the results for the 6 degrees of freedom (6DOF) environment are presented. As in chapter 5 two training runs are shown: the first for a set of simplified initial conditions (lower height, lower velocity, velocity directed only downwards), then a set of initial conditions sourced from historic flight data of the Falcon 9 [40] is analyzed. This second set of data simulates landing on a downrange location, such as a barge in the middle of the ocean or a downrange landing pad.

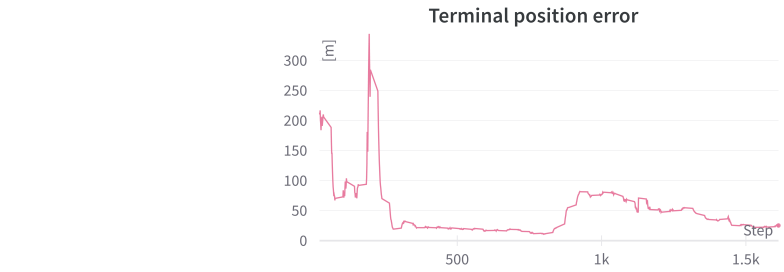
6.1 Simplified initial conditions

The first case studied is the one of simplified initial conditions: the launcher starts from a lower height and with a lower downward-only *mean* velocity. The vector of mean initial conditions $\boldsymbol{\mu}_{\boldsymbol{x}_0}$ and its range $\boldsymbol{\Delta}_{\boldsymbol{x}_0}$ are reported in Tab.4.5.

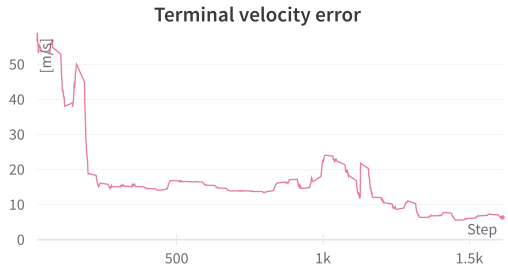
In this case the *policy entropy coefficient* in 2.27 is set to $c_{\text{entropy}} = 0.01$ to encourage exploration and avoid premature convergence of the policy to a local optimum. As can be seen in Fig. 6.1 the algorithm first converges to a model in the 1st training phase, then around time step 800 the change in reward function kicks off a second exploratory phase, resulting in convergence to a more efficient (in terms of propellant used) policy with a lower terminal velocity.

The lower consumption of propellant is also highlighted by the sharp reduction in average episodic length, as shown in Fig. 6.7b.

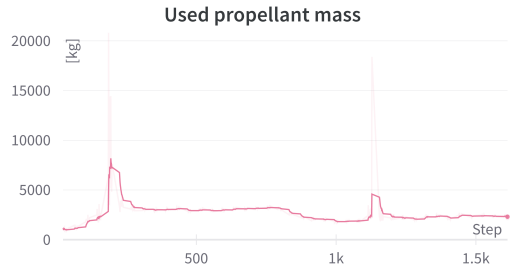
The behavior of some metrics of the PPO algorithm can be assessed to verify convergence from an algorithmic perspective. In particular the mean reward has an increasing trend, with a dip at training step 800 due to the switch in reward functions (this implies that there will be a different maximum achievable reward) and the explained variance converging to a value of 1 mean shows a good approximation of the value function by the value network.



(a) $||\boldsymbol{r}_f||$



(b) $||\boldsymbol{v}_f||$



(c) $m(t_f) - m(t_0)$

Figure 6.1: Trend of trajectory metrics during training, showing successful convergence.

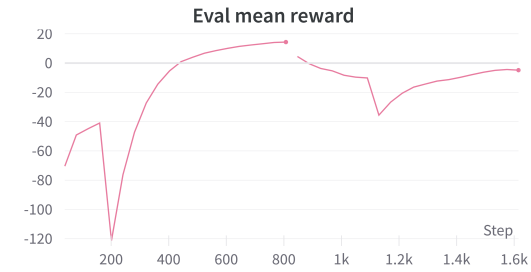
6.1.1 Montecarlo analysis of the policy

In order to test the robustness of the policy a Montecarlo analysis is carried out. In this run, since two different reward functions have been used for the training phase, it is interesting to compare the difference between the best model obtained during the 1st phase (the one with the shaped reward) and the model obtained after the 2nd phase of training (without the target velocity reward). The Montecarlo analysis is performed on mean initial conditions and range of initial conditions as shown in Tab.6.3.

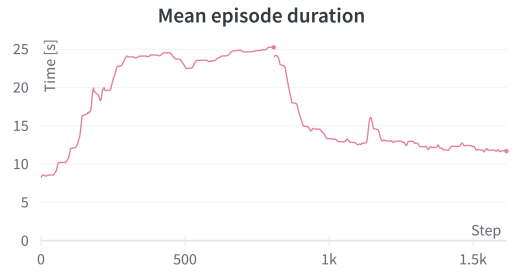
The distribution of position, velocity, attitude and angular velocity terminal errors can be seen in Fig. 6.3a, b, c and d for the first phase and in Fig. 6.4a, b, c and d for the second, along with the distributions of the used propellant mass (Fig. 6.3e, Fig. 6.4e) and the terminal velocity angle (Fig. 6.3f, Fig. 6.4f), computed as Eq. 6.1. This angle measures the deviation of the terminal velocity from the vertical at touchdown, which would nominally be directed downwards to prevent the lander from tipping over.

$$\phi = 180 - \arcsin\left(\frac{v_x(t_f)}{||\boldsymbol{v}(t_f)||}\right) \tag{6.1}$$

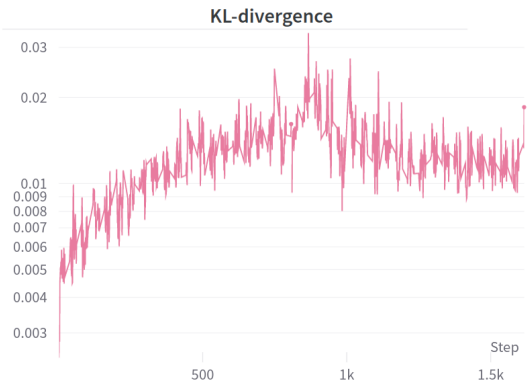
An approximation of the *probability distribution function* (*pdf*) is overlapped to the histograms, computed using a kernel density estimation with Gaussian kernels



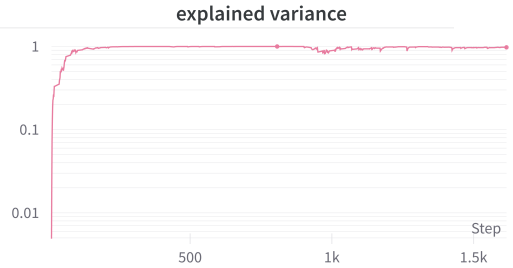
(a) Evaluation mean episodic reward



(b) Mean episodic length



(c) Approximate KL-divergence



(d) Training explained variance

Figure 6.2: Reinforcement Learning training metrics (simplified initial conditions). In particular, the downward trend in episode duration around 10^3 steps shows the strong minimization of fuel consumption when the second reward phase begins.

and automatic bandwidth determination (implemented by the *Pandas* library).

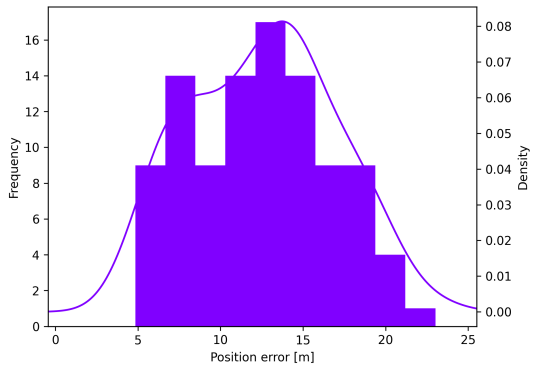
The statistic for the terminal errors of phase 1 and phase 2 are reported in Tab.6.1 and Tab.6.2.

	Position	Velocity	Attitude	Angular velocity	Used mass	Velocity angle
μ	$12.5\,m$	$13.9\,m/s$	$4.3°$	$0.01°/s$	$3214\,kg$	$4.6°$
σ	$4.3\,m$	$0.3\,m/s$	$2.7°$	$0.01°/s$	$96\,kg$	$1.4°$

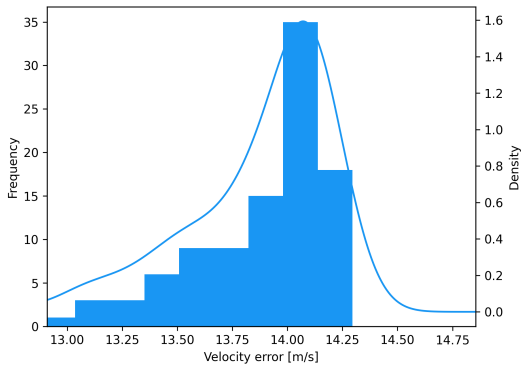
Table 6.1: Terminal errors and used mass statistics, 1st phase (mean μ and standard deviation σ). While the position error is reasonably low, the velocity error is slightly too high at the end of this phase.

	Position	Velocity	Attitude	Angular velocity	Used mass	Velocity angle
μ	$25.1\,m$	$6.78\,m/s$	$3.6°$	$0.03°/s$	$2467\,kg$	$48°$
σ	$3.5\,m$	$1.2\,m/s$	$2.0°$	$0.0°/s$	$88.4\,kg$	$9.9°$

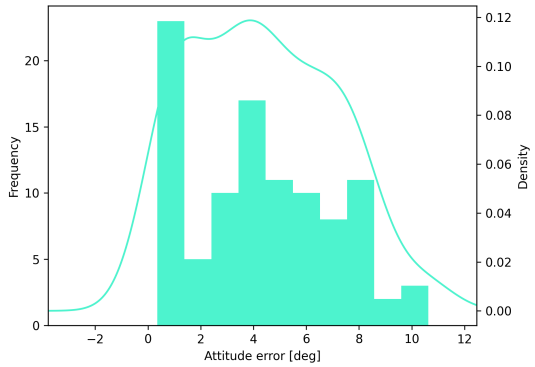
Table 6.2: Terminal errors and used mass statistics, 2nd phase. The position and velocity errors have reasonable values, however the velocity angle at touchdown becomes high, meaning that there is a significant horizontal component of the velocity.



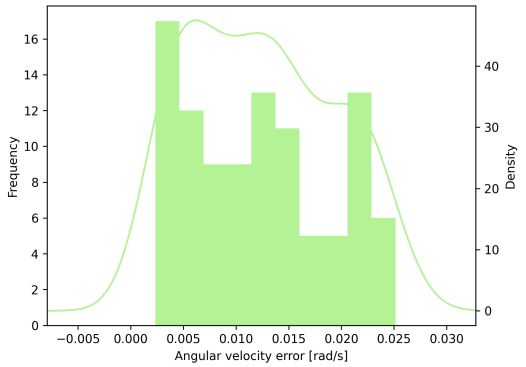
(a) Position error



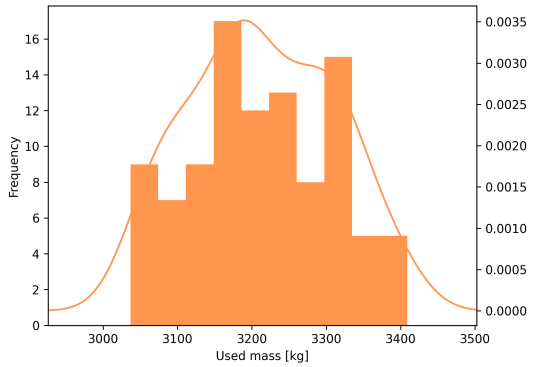
(b) Velocity error



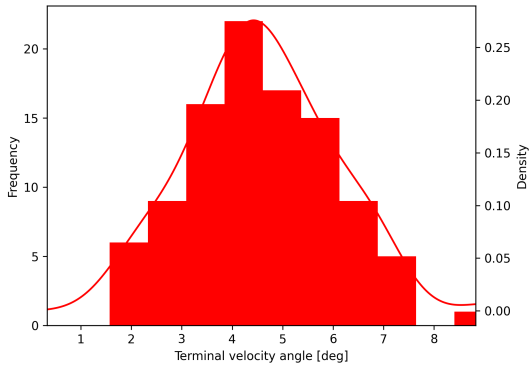
(c) Attitude error



(d) Angular velocity error



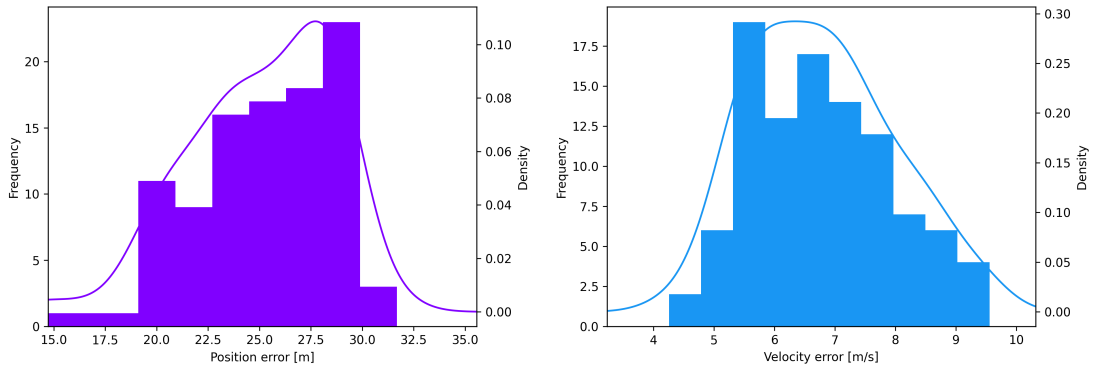
(e) Used mass



(f) Terminal velocity angle

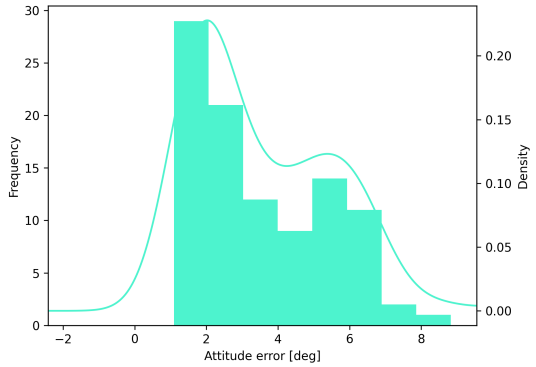
Figure 6.3: Montecarlo analysis distribution of terminal errors with the best model in the **1st training phase** (simplified IC). The errors show a mono modal distribution, however the terminal velocity is a bit high to have a soft touchdown.

It is quite interesting to analyze the differences between the two controllers. Comparing the position errors (Fig. 6.3a,Fig. 6.4a), the velocity errors (Fig.

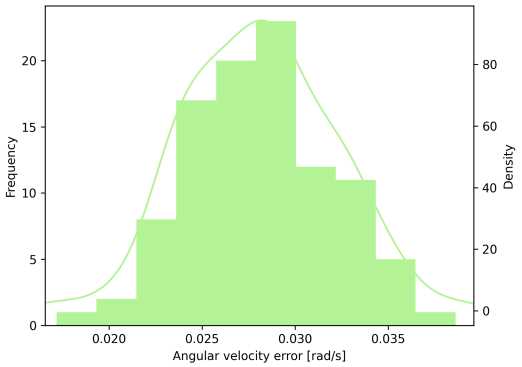


(a) Position error

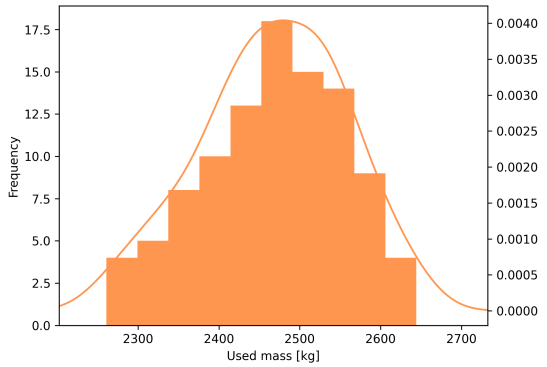
(b) Velocity error



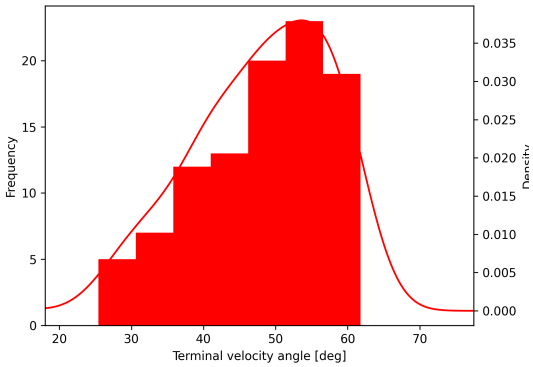
(c) Attitude error



(d) Angular velocity error



(e) Used mass



(f) Terminal velocity angle

Figure 6.4: Montecarlo analysis distribution of terminal errors with the best model in the **2nd training phase** (simplified IC). The velocity error decreases with respect to the first phase, but there is an increase in position error. The used propellant mass is sharply reduced with respect to the previous phase.

6.3b,Fig. 6.4b) and the used mass (Fig. 6.3e,Fig. 6.4e) it's clear that the first controller aims to minimize the terminal position error while having a relatively high velocity error and propellant consumption, while the second controller minimizes successfully the used propellant mass (using about 30% less propellant), lowering at the same time the average velocity error to about $6.5\,m/s$, down from about $14\,m/s$. This is however detrimental to the position error which grows from an average of $14\,m$ to about $26\,m$. Furthermore, the terminal velocity angle increases to about $4°$ to an average value of $50°$, although given the low terminal velocity this might not affect the stability of the lander. Both attitude and angular velocity errors have comparable statistics across the two training phases and are within the acceptable bounds for the landing.

A sample trajectory from the second model is shown in Fig. 6.5. The lander first rotates to thrust in the landing direction, then when a sufficiently low altitude is reached the thruster performs a burn to reduce the velocity and finally once over the landing pad a vertical attitude is reached and successful landing is achieved.

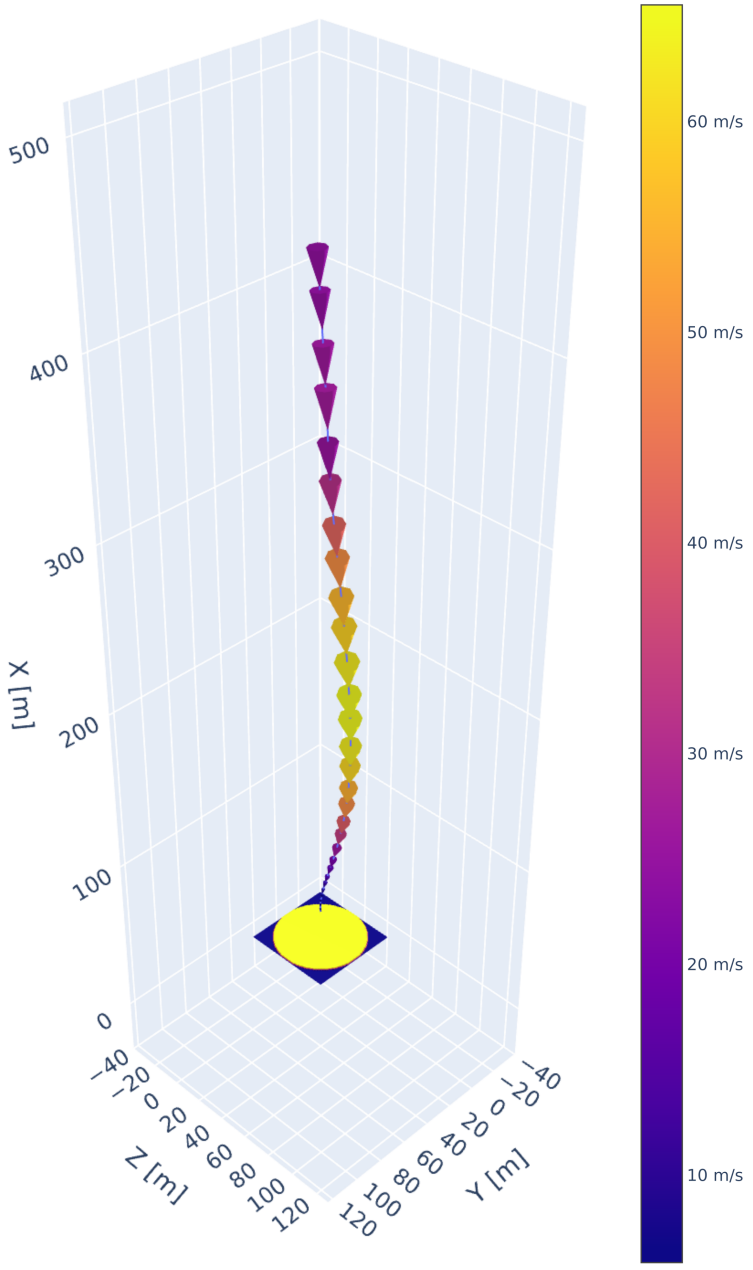


Figure 6.5: Trajectory of an episode using the optimal network. We can notice the lander accelerating due to gravity in the first part of the trajectory, reaching a maximum in velocity around $200m$ of height. It then rapidly decelerates to achieve a soft landing with minimal fuel consumption.

6.2 Realistic initial conditions

In the case of realistic initial conditions the vector of mean initial conditions of the state $\boldsymbol{\mu}_{\boldsymbol{x}_0}$ and its range of dispersion $\boldsymbol{\Delta}_{\boldsymbol{x}_0}$ are reported in Tab.4.7.

In this case two important hyperparameters to tweak were the *batch size* and the *number of steps per rollout*. Due to the increased length of the average episode and the size of the action space these had to be increased significantly, to have a larger number of samples available, capable of stabilizing the update of the policy and value networks.

In order to pick a robust policy a periodic evaluation of its behavior is performed, running it for 15 episodes and computing the mean reward. The highest mean reward will correspond to the lowest final position and velocity errors, due to the high terminal bonus given based on them, and thus to the best overall policy.

Convergence requires significant more episodes, clocking in at around 24h of runtime, due to the need for the algorithm to explore a much larger action and observation space.

The convergence behavior of the algorithm can be analyzed in Fig. 6.7 and Fig. 6.6. In Fig. 6.6 a more detailed interpretation of the convergence behavior can be drawn. Comparing the mean episodic reward Fig. 6.7a with the other metrics it is evident that the algorithm first explores tracking the target acceleration, then around step 1500 it discovers the terminal bonus, successfully landing with low speed and position errors. Then there is an exploratory phase in which it attempts to further optimize the reward, finally converging to a solution with higher terminal reward due to the decrease in used mass.

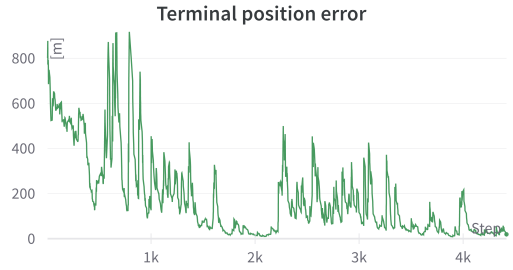
The robustness of the PPO algorithm is evident in the trends of its training metrics Fig. 6.7. The explained variance, measuring the accuracy of the value function in predicting the cumulative rewards reaches almost 100%, showing how the reward landscape is thoroughly sampled and quantified.

It can be seen that the behavior is noisy and that there are spikes in the terminal errors. This could be caused both by an excessive residual degree of exploration or by the need for the policy to be more robust, and could benefit from further increasing the batch and sample sizes.

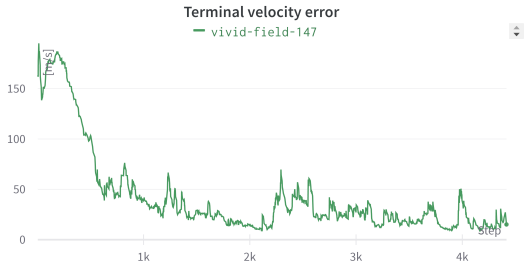
It is possible to visualize the trajectory of an episode with good landing behavior Fig. 6.8. It can be seen that the rocket manages to reach successfully the landing zone (yellow circle in the figure) with low terminal velocity and a vertical profile for the terminal descent.

6.2.1 Montecarlo analysis of the policy

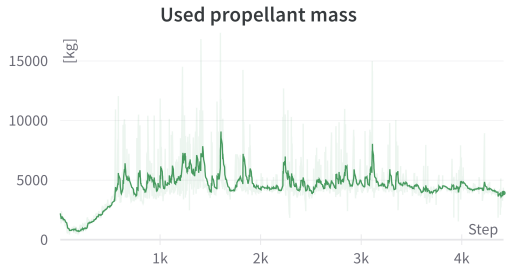
In order to test the robustness of the policy a Montecarlo analysis is carried out on the best-performing policy network. The Montecarlo is performed on mean initial conditions and range of initial conditions as shown in Table 6.3.



(a) $||\boldsymbol{r}_f||$



(b) $||\boldsymbol{v}_f||$



(c) $m(t_f) - m(t_0)$

Figure 6.6: Trajectory metrics trend in the case of realistic initial conditions. There is high exploration, so the need to isolate a good policy arises. This is achieved by periodically evaluating the agent and saving the policy with the highest mean reward.

	$\boldsymbol{r}$	$\boldsymbol{v}$	$\boldsymbol{q}$	$\boldsymbol{\omega}$	m
$\boldsymbol{\mu}_{\boldsymbol{x}_0}$	$[2000, -1600, 0]$	$[-90, 180, 0]$	$[0.866, 0, 0, -0.5]$	$[0, 0, 0]$	$41e3$
$\boldsymbol{\Delta}_{\boldsymbol{x}_0}$	$[100, 200, 50]$	$[30, 30, 10]$	$[0.1, 0.1, 0.1, 0.1]$	$[0.05, 0.05, 0.05]$	$1e3$

Table 6.3: Mean and range of initial conditions in the case of Monte Carlo analysis.

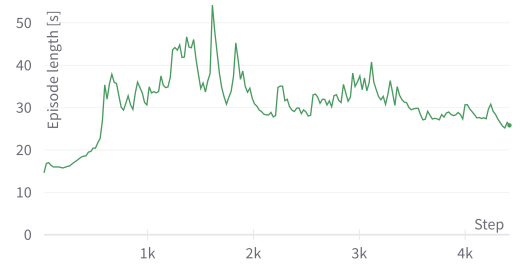
The histogram of the distribution of the terminal errors can be seen in Fig. 6.10.

The statistic for the terminal errors are reported in Tab.6.4.

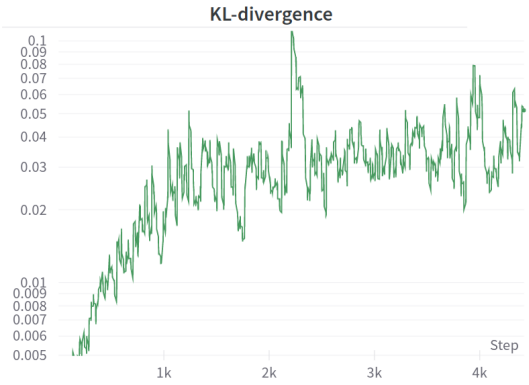
The performance of the controller is compared to a baseline obtained by solving the fuel-optimal problem disregarding the rotational dynamics of the problem, meaning that the body is treated as a point mass and the constrained (null terminal velocity and distance from landing pad) optimization problem is solved. In this context 3DOF is intended as translational degrees of freedom only (differently than in the previous chapters).

The RL solution is also compared to a *successive convexification MPC* approach. The results, reported in Table 6.5 highlight that the optimal solution is about 25% more efficient than the Reinforcement Learning one. This is due to both the acceleration-tracking shaping and the need to perform attitude control

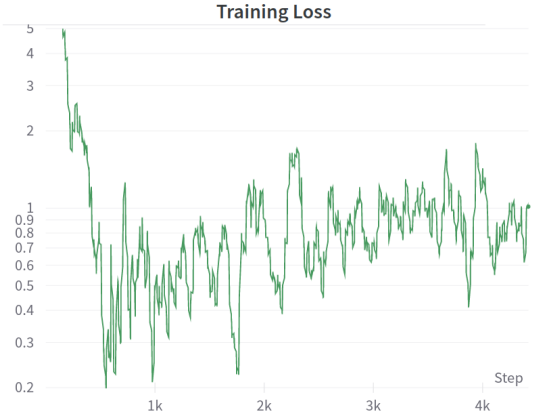
(a) Evaluation mean episodic reward



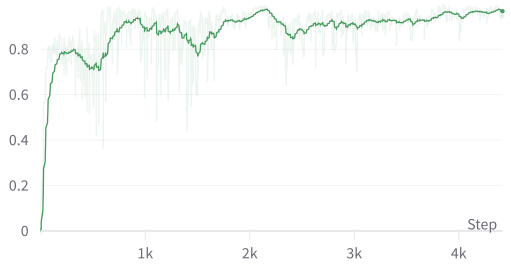
(b) Evaluation mean episodic length



(c) Approximate KL-divergence



(d) Training loss L



(e) Training explained variance

Figure 6.7: Reinforcement learning training metrics. The upwards trend in episodic reward show successful learning of a good policy. The decrease in training loss also signals convergence, as well as explained variance showing that the agent has explored the environment.

as well. In fact the RL controller requires only about 60% of the propellant used by the successive convexification solution, thus being significantly more efficient.

	Position	Velocity	Attitude	Angular velocity	Used mass	Final velocity angle
μ	$10.0\,m$	$9.42\,m/s$	$4.7°$	$0.06°/s$	$4219\,kg$	$18.9°$
σ	$4.3\,m$	$2.3\,m/s$	$2.7°$	$0.04°/s$	$4219\,kg$	$9.5°$

Table 6.4: Terminal errors and used mass statistics

	RL controller (6DOF)	3DOF optimal solution	SCVX (6DOF)
$\lvert m_0 - m_f \rvert$	$4250\,kg$	$3545\,kg$	$7525\,kg$

Table 6.5: Propellant consumption comparison of the obtained policy, the 3DOF (point mass) optimal solution and 6DOF successive convexification (SCVX) approaches

6.3 Robustness to unmodeled dynamics and disturbances

The policy obtained is analyzed to quantify its robustness to unmodeled dynamics effect and external disturbances. These are:

- **Error in the position of the CoM**
- **Flexible modes of the structure**
- **Uncertainty in the inertia moments**
- **Real dynamics of the actuators**
- **Misalignment of the thrust**
- **Wind gusts**
- **Wind layers model**

Error in the position of the CoM

The center of mass' position of the dry vehicle $x_{\text{CG}}^{\text{dry}}$ is taken to be within a range of $\pm 3\%$ of the nominal position of $15m$.

Furthermore, the shift of the center of mass due to propellant consumption has been modeled. It is computed by taking into account the variation in propellant mass split between the oxidizer and fuel through the mixture ratio parameter O/F.

The center of gravity location (CG) is computed at each instant through Eq.6.2

$$x_{\text{CG}} = \frac{m_{dry} x_{\text{CG}}^{\text{dry}} + m_{ox} x_{\text{CG}}^{\text{ox}} + m_{fuel} x_{\text{CG}}^{\text{fuel}}}{m_{dry} + m_{ox} + m_{fuel}} \tag{6.2}$$

The position of the center of mass of both fuel ($x_{\text{CG}}^{\text{fuel}}$) and oxidizer ($x_{\text{CG}}^{\text{ox}}$) and their masses ($m_{\text{ox}}$ and m_{fuel}) are *time-varying* due to their consumption, with their behavior controlled by the change in propellant mass and the mixture ratio, as described in equation 6.4 and 6.3 respectively.

$$\begin{aligned} m_{\text{prop}} &= m(t) - m_{\text{dry}} \\ m_{\text{fuel}} &= \frac{m_{\text{prop}}}{1 + O/F} \\ m_{\text{ox}} &= m_{\text{prop}} \frac{O/F}{1 + O/F} \end{aligned} \tag{6.3}$$

$$\begin{aligned} x_{\text{CG}}^{\text{fu}} &= x_{\text{CG}}^{\text{fu}}(t_0) \frac{m_{fuel}(t)}{m_{fuel}(t_0)} \\ x_{\text{CG}}^{\text{ox}} &= h_{\text{tank}} + (x_{\text{CG}}^{\text{ox}}(t_0) - h_{\text{tank}}) \frac{m_{fuel}(t)}{m_{fuel}(t_0)} \end{aligned} \tag{6.4}$$

Flexible modes of the structure

To model the dynamics and kinematics of the lander the rigid body assumption is made. To analyze the effects of flexibility due to the flexing modes of the structure a perturbing force and torque are introduced. The dynamics of these modes is computed and the values of force and torque are fed to the rigid-body simulator.

To model the flexural dynamics of the vehicle the modal variables q_i about the y and z axes are integrated from Eq.6.5, with ω_i being the eigenfrequency associated to each $i-th$ mode, ξ its damping coefficient and $t_{p,i}$ a structural parameter.

$$\begin{aligned} \ddot{q_i^y} &= -\omega_i^2 q_i^y - 2\xi\omega_i \dot{q}_i^y - T_y t_{p,i} \\ \ddot{q_i^z} &= -\omega_i^2 q_i^z - 2\xi\omega_i \dot{q}_i^z - T_z t_{p,i} \end{aligned} \tag{6.5}$$

The forces and torques about the z_B and y_B body axes are computed respectively as 6.6 and 6.7, being x_{CG} the longitudinal position of the center of mass and $r_{p,i}$ a structural parameter.

$$\begin{aligned} F_{\text{flex}}^y &= T_y \sum_i r_{p,i} q_i^y \\ F_{\text{flex}}^z &= T_z \sum_i r_{p,i} q_i^z \end{aligned} \tag{6.6}$$

$$\begin{aligned} M_{\text{flex}}^y &= -T_z \sum_i (r_{p,i} x_{CG} + t_{p,i}) q_i^z \\ M_{\text{flex}}^z &= T_y \sum_i (r_{p,i} x_{CG} + t_{p,i}) q_i^z \end{aligned} \tag{6.7}$$

Uncertainty in the inertia moments

While in training the inertia matrix is assumed to be constant throughout each episode (being computed at the beginning using the initial mass), in the sensitivity analysis it is recomputed at each time step to account for the change in propellant mass, thus being time-varying.

The inertia moments are going to have some uncertainty, so an error in the range of 1% is sampled from a uniform distribution and multiplies the time-varying inertia moments I_{xx}, I_{yy}, I_{zz}.

Real dynamics of the actuators

The actuators will have their own dynamics that will result in delays between the commanded variables (denoted by the superscript $\cdot^{\text{cmd}}$) and their actual output.

The TVC gimbal actuators are modeled as second order low pass filters, with ω_{act} and ξ_{act} respectively the natural frequency and the damping coefficient of the actuator. The transfer function from the commanded gimbal angle δ_i^{cmd} and the output one is 6.8.

$$\frac{\delta_i}{\delta_i^{cmd}}(s) = \frac{\omega_{\text{act}}^2}{s^2 + 2\xi_{\text{act}}\omega_{\text{act}} + \omega_{\text{act}}^2} \tag{6.8}$$

Furthermore, the thruster will have a certain delay in producing the required thrust [45] and has been modeled as a first order low pass filter with characteristic time $\tau_{\text{thrust}} = 2s$, and thus natural frequency $\omega_{\text{thrust}} = \frac{2\pi}{\tau_{\text{thrust}}}$. Its dynamic response is shaped by equation 6.9.

$$\frac{||\boldsymbol{T}||}{||\boldsymbol{T}_{cmd}||}(s) = \frac{\omega_{\text{thrust}}}{\omega_{\text{thrust}} + s} \tag{6.9}$$

Misalignment of the thrust

Slight misalignment of the thruster can be expected in real-life launch vehicles. To account for this the thrust vector is multiplied by a rotation matrix R_ϵ modeling a slight offset ϵ_i around each of the three axis, with $i \in x, y, z$:

$$R_\epsilon = \begin{bmatrix} \cos\epsilon_y \cos\epsilon_z & \cos\epsilon_y \sin\epsilon_z & -\sin\epsilon_y \\ -\cos\epsilon_x \sin\epsilon_z + \sin\epsilon_x \sin\epsilon_y \cos\epsilon_z & \cos\epsilon_x \cos\epsilon_z + \sin\epsilon_x \sin\epsilon_y \sin\epsilon_z & \cos\epsilon_y \sin\epsilon_x \\ \sin\epsilon_x \sin\epsilon_y + \cos\epsilon_x \sin\epsilon_y \cos\epsilon_z & -\sin\epsilon_x \cos\epsilon_z + \cos\epsilon_x \sin\epsilon_y \sin\epsilon_z & \cos\epsilon_x \cos\epsilon_y \end{bmatrix} \tag{6.10}$$

The misalignment angles are sampled from a uniform distribution with zero mean and range $\epsilon_i \in [-0.5°, 0.5°]$.

Wind gusts

Wind gusts are a relevant disturbance in the considered height range, due to phenomena such as thermal inversion, down wash etc. A simple cosinusoidal model has been employed, with the values for the magnitude and height range of the gusts sourced from [46].

They're modeled as sinusoidal gusts with an amplitude specified by the vector $\boldsymbol{A}_{\text{gust}} = [15, 15, 15]\ m/s$. In the simulation each gust is taken to have a height of $\Delta h = 100m$, between $h_1 = 1.5$ and $h_2 = 1.6km$, and they're computed as:

$$\boldsymbol{V}_{\text{gust}} = \boldsymbol{A}_{\text{gust}} \left(1 - \cos\left(\frac{\pi(x - h_1)}{0.5\Delta h}\right)\right) \tag{6.11}$$

Wind layers model

To implement zonal and meridional winds, the *Horizontal Wind Model 14* block of Simulink has been used, implementing the U.S. Naval Research Laboratory model.

6.3.1 Sensitivity results

In some episodes the control action was unable to make the trajectory converge to the landing site. These runs, defined as *outliers*, were excluded from the statistic as they would otherwise make the mean and standard deviation lose relevance, and their number is reported in Tab.6.6. To understand better the reason for the divergence, these outlier trajectories are analyzed, and a pattern is noticed: in all of them the vehicle first gets to the landing site with very high accuracy (a position error in the order of a few meters) and then the vertical velocity reaches zero when it is a few meters above the pad. The control action then keeps the rocket hovering or propels it upwards, making it land far away in an uncontrolled manner. This behavior could be addressed either during training by modifying the reward function to penalize an upwards velocity below a certain threshold or by employing a landing mode controller when the rocket reaches a waypoint a few meters above the surface.

Overall the controller is robust to the uncertainties in the specified ranges, having position and velocity errors in ranges comparable to the ones without these effects, as shown in Fig. 6.11 and compatible with a successful landing, as can be seen in the dispersion plot, Fig. 6.12.

Disturbance	outliers
Center of mass error	0
flexible modes	0
inertia moments error	1
offset CoM	1
real actuator dynamics	2
thrust misalignment	0
wind gusts	0
wind layers model	1

Table 6.6: Number of outliers, runs where there is divergence of the controller, out of 100 runs for each disturbance.

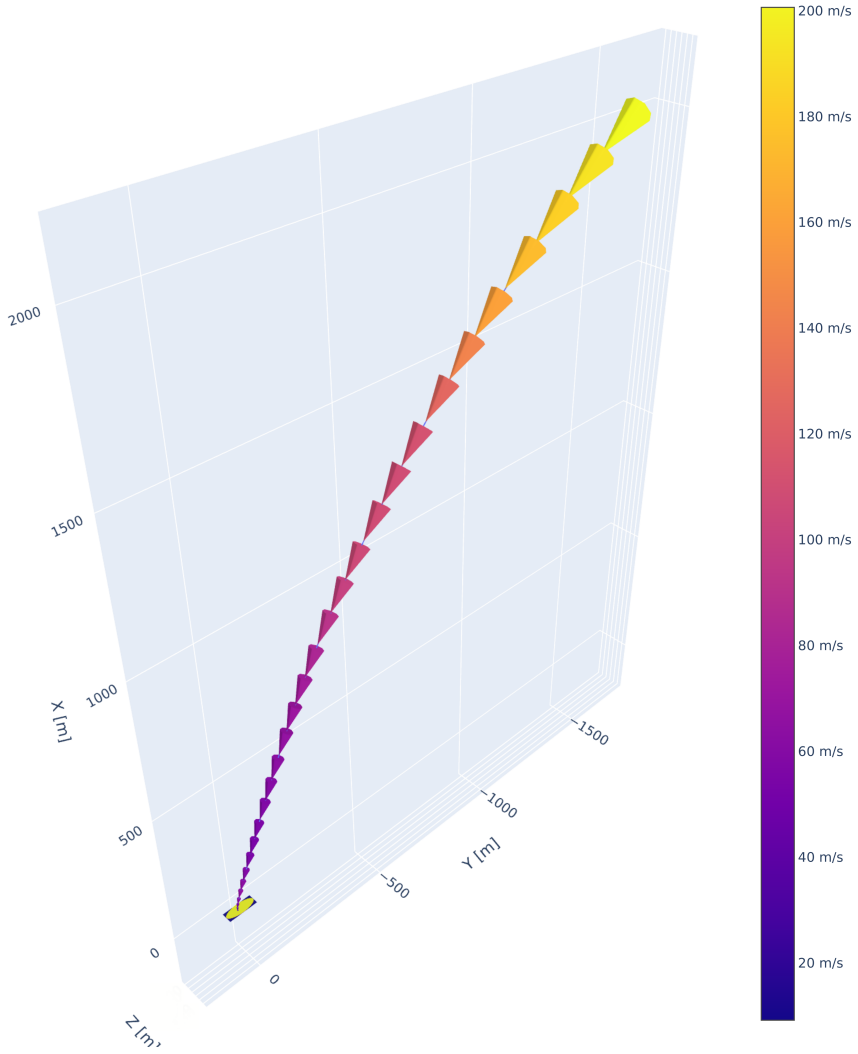


Figure 6.8: Trajectory of an episode using the optimal network. We can notice the decrease in velocity and the successful pinpoint landing.

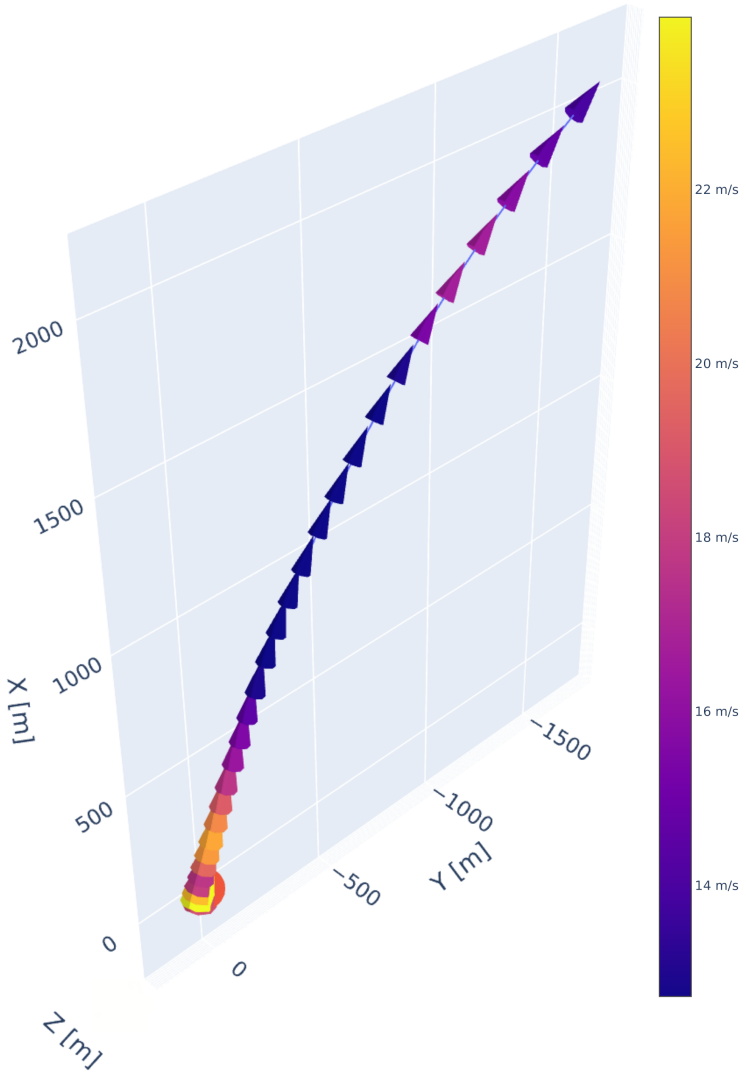
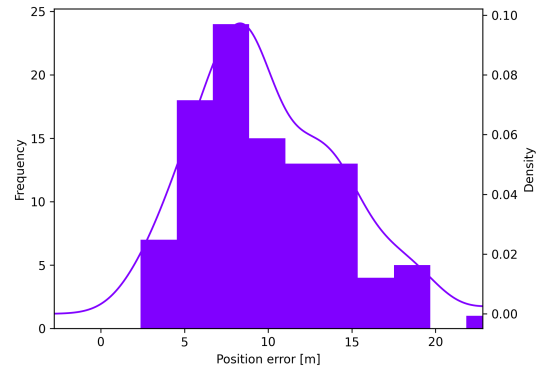
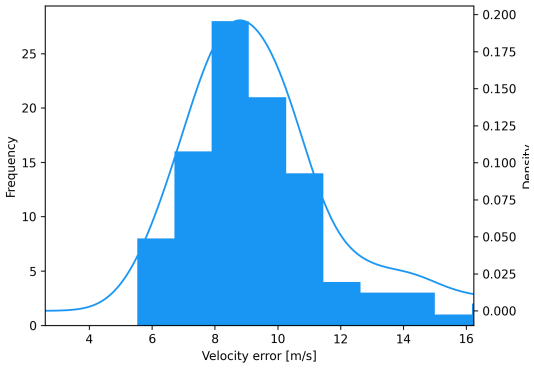


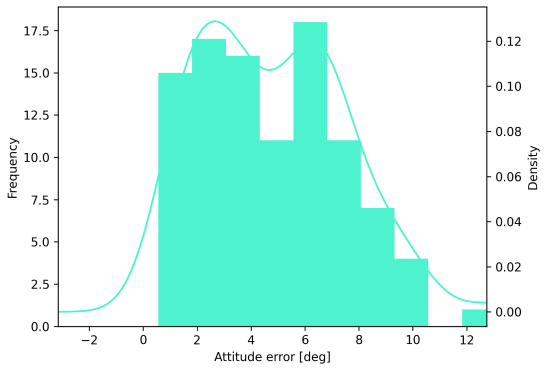
Figure 6.9: Target acceleration along the trajectory. We can notice at the end a high target velocity due to the proximity to the landing site. The agent learns to not strictly follow this command as it would require leaning excessively.



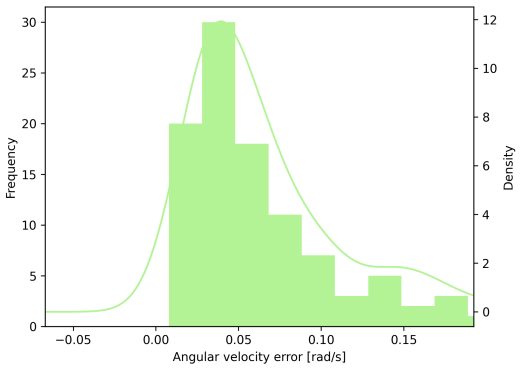
(a) Position error



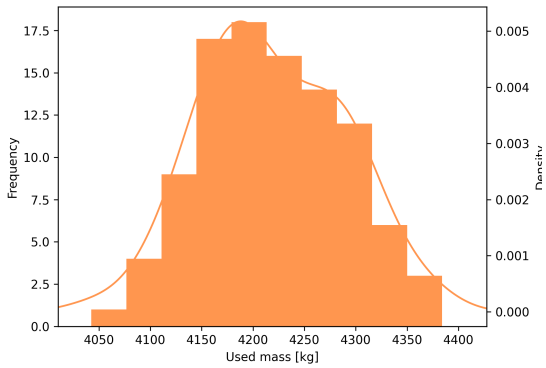
(b) velocity error



(c) attitude error

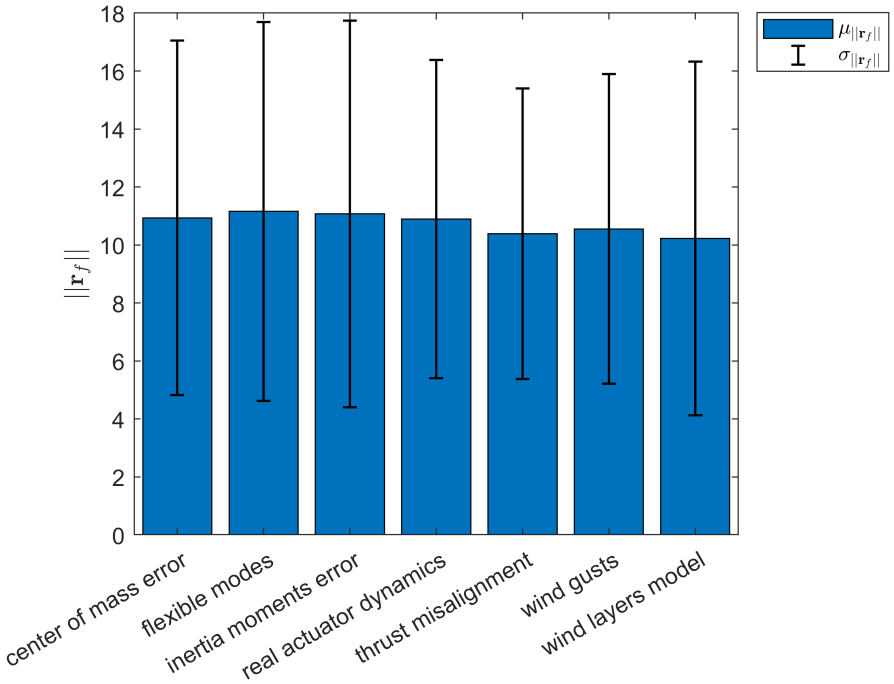


(d) angular velocity error

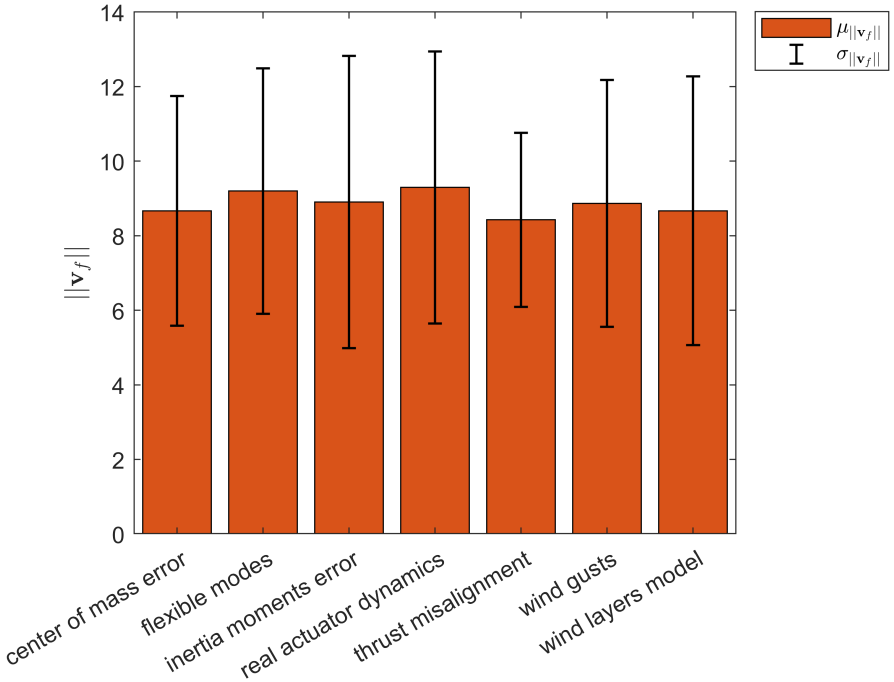


(e) used mass

Figure 6.10: Montecarlo analysis distribution of terminal errors, showing monomodal distribution of the errors. Low position error and terminal velocity make this compatible with soft landing.



(a) Final position error



(b) Final velocity error

Figure 6.11: Sensitivity to unmodeled dynamics and disturbances. The policy proves to be robust by having low errors and dispersion in the presence of different disturbances and uncertainties.

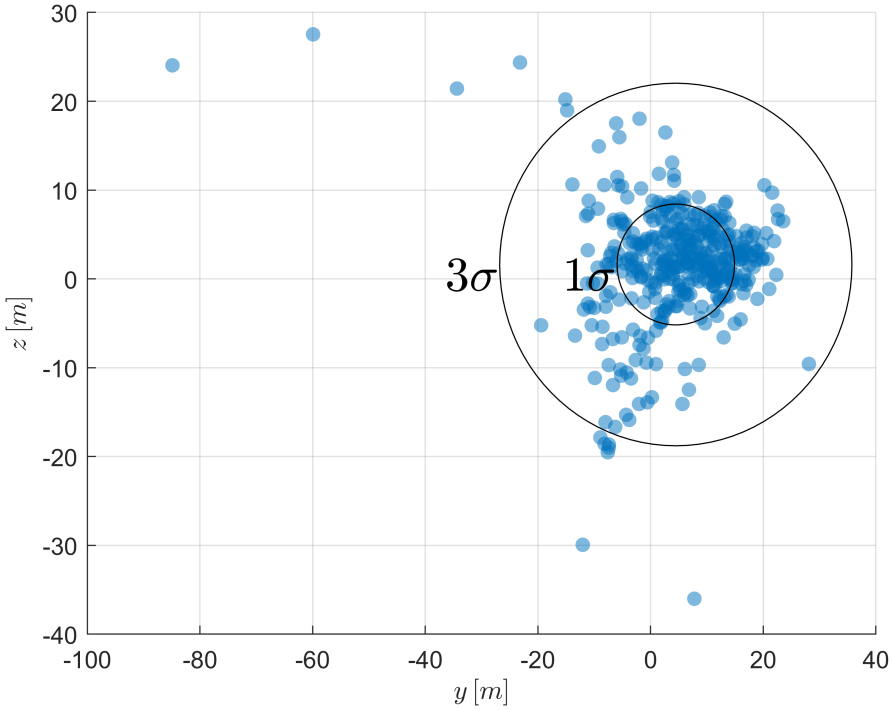


Figure 6.12: Dispersion plot with disturbances and unmodeled dynamics. Most of the simulations result in a successful landing within a $20m$ radius, however there are a few significant outliers.

Chapter 7

Conclusions

In this thesis a Reinforcement Learning controller is developed for the task of landing a reusable launcher's first stage and its robustness is quantified. Several aspects of this novel control technique are analyzed, starting from how parameters selection affects convergence behavior and how the reward and activation functions influence performance.

There are three main contributions made with this thesis, following the objectives in 1.1.1:

- A model-free RL algorithm (PPO) is applied to develop an integrated G&C controller. The effects of different reward functions, hyperparameters and network structures are assessed and quantified. An important novel aspect is the assessment of the robustness of the obtained control policy to unmodeled dynamics and parametric uncertainties not present during training.

- A modular and easily expandable nonlinear 3DOF and 6DOF simulators for launchers/landers control is developed, compatible with standard RL frameworks and validated. This simulator is publicly released [1] to enable researchers to use a standardized and validated environment in the future.

- An easy-to-use pipeline for cloud-accelerated RL training is created, integrated with web-based metrics visualization tools. This allows to scale the training algorithm in the future to speed up its run time and to easily interpret and compare the results of each training run.

7.1 Achieved objectives

Simulation environment development Two different simulators and environments with *de-facto* standard OpenAI Gym APIs compatibility have been developed.

[1] Available on PyPi `https://pypi.org/`

A 6DOF environment, validated on the Simulink simulator developed in [9], featuring RK45 integration of the equations of motion, realistic 3D rendering, realistic lower ISA atmospheric model (easily extensible to the upper layers to simulate the full mission profile).

A 3DOF environment (two translational degrees of freedom plus one rotational) was developed and used to inform the choice of reward functions for the 6DOF environment and to familiarize with the problem.

Two key elements of the environment that proved key in achieving convergence are *normalizing* both the action and the observations it gets respectively as input and output, and to set appropriate *termination conditions* to terminate the episodes after a certain amount of time or when the agent gets outside certain bounds.

Reinforcement learning algorithm Both environments have been trained using the *PPO* (Proximal Policy optimization) algorithm, which suits best the problem for its ability to use continuous action and state/observation spaces and its high sample efficiency.

This algorithm shapes a *policy neural network* which outputs an action to execute for each observation it gets. The aim is to shape the neural network as to maximize the total discounted reward over episodes, given by a reward function.

The PPO algorithm achieves successfully convergence to a good policy, however some tweaking of hyperparameters was necessary. In particular the batch size of state transitions used for the optimization steps was found to be key in obtaining low terminal speed and position errors. Sensitivity with respect to this parameter was high in the 6DOF case, while being low in the simpler 3DOF case. Being aware of the shape of the reward function was important to inform the selection.

Different topologies and activation functions for the networks have been tested. The activation function has shown to have a key role on the convergence' behavior, however once convergence is achieved, performance in terms of mean episodic reward was similar. The topology of the network was also influential: while at first a small network architecture (two hidden layers with 64 neurons each) was employed and obtained good performance in the 3DOF case, moving to a 6DOF scenario caused it to reach the limit of its learning abilities, resulting in poor performance. Doubling the first layer unlocked the ability to learn better policies, however, doubling also the second hidden layer or adding a third layer resulted again in poor performance. We hypothesize that this is due to the difficulty of training larger networks due to increased number of parameters.

Experiment setup and execution pipeline The results in both environment are presented, both starting from two different sets of initial conditions:

- A simplified set, starting from a lower altitude and lower initial velocity,

with the latter being *nominally vertical*

- A Realistic set, using a range of initial conditions sourced from historic flight data of the Falcon 9 launch vehicle. This set is representative of real-world conditions for the terminal landing burn of a modern 2-stages launch vehicle.

Training is performed on Virtual Machines on the cloud, through a Continuous Deployment pipeline that allows to quickly deploy several revisions of the reward function to make experiments. This has proved critical to properly shape the reward function in a reasonable time and to achieve reasonable run times for training. Indeed, without this setup, training would have taken a full day (at least) on local machines, dramatically slowing down the investigation on the effects of the hyperparameters and the reward function.

Reward function development and analysis Two different reward functions have been employed:

- The first one, denominated *target-velocity based*, follows a (heuristic) target velocity aiming in the direction of the Line-Of-Sight (LOS) to the landing target, with magnitude decreasing exponentially as the lander approaches ground. The lander also obtains a terminal bonus for achieving landing successfully. This reward function has been employed in a two-phase approach: once the policy consistently achieves landing, the reward function is *ablated*, with the target-velocity deviation penalty term being dropped. This approach, that we named *reward annealing*, should encourage the agent to minimize fuel consumption while still achieving the terminal *landing bonus* discovered in the first exploratory phase. In the case of the simplified initial conditions this is achieved, while it is not successful for the realistic ones. Obtaining a policy that achieves, in the first phase of training, the terminal bonus for successful landing more consistently could solve this problem by better engraining the correct behavior in the policy before the heuristic target velocity is removed from the reward signal.

- The second reward function, denominated *target-acceleration based* developed aims to closely track a target acceleration, which is computed as the solution of the simplified (no atmospheric effects) 3DOF-translational landing problem that minimizes the integral of the square of the acceleration over the trajectory. This target acceleration should act as a proxy to *hint* the thrust-optimal trajectory to the agent, with further reward terms that incentivize the agent to minimize propellant consumption. This approach is more robust than the target-velocity reward and achieves a successful landing also in the 6DOF environment with realistic initial conditions. However, this approach could be pushing the agent excessively to aim for a control action that tracks the target acceleration, not letting it explore more fuel-efficient trajectories. In other words the agent could learn to be too greedy.

Both reward functions have in common the presence of penalty terms to penalize exceeding attitude bounds and a penalty for the usage of the thruster, aiming to minimize the propellant mass used to attempt a hover-slam maneuver.

Also present in both is a significant *landing bonus* term, that informs the agent of the achievement of a successful landing, defined in terms of position, velocity and attitude errors at touchdown.

The two different reward functions, developed in detail in 4.5, have different trade-offs, specializing better to different initial conditions. The two-step approach tested has proven promising in terms of optimization, but improvements on its convergence behavior are needed to expand its generalization capabilities, as it fails to work in more challenging initial conditions. One reason for this could be that the first phase of training would need to run longer, to have the policy learn how to *consistently* obtain the terminal landing bonus, which is the main reward term remaining in phase two.

Evaluation of different activation functions The different activation functions, hyperbolic tangent ($tanh$) and Rectified Linear Unit ($ReLU$), tested in 2.3.4 show trade-offs between speed of convergence and stability, with the first converging more consistently but also more slowly, and the second showing a faster convergence but with an un-learning behavior (the network tends to forget the learned weights after reaching a quasi-optimal policy, leading to a decay of the mean episodic return). A study on the hyperparameters that affect these two aspects would help clarify if the increased speed of convergence of the ReLU function can be exploited more consistently, without having to limit its rate of convergence in order to prevent the un-learning behavior.

Exploration of hyperparameters space The hyperparameters of the algorithm have proven important in convergence of the policy. The default hyperparameters' values of [42] have proven to be mostly appropriate, however an increased number of transitions in the replay buffer and a higher number of episodes before each policy update phase turned out to be critical for the algorithm to stabilize the direction of update of the parameters and consistently maximize the episodic return. We hypothesize that this is due to the sparse reward landscape (the terminal bonus is given only on the final transition), which means that a low number of transitions cannot capture the final bonus consistently, leading to the update step getting stuck in local minima.

Policy performance The policies developed for both sets of initial conditions and both environments reach a satisfactory performance, compatible with a successful landing. The performance is better for the simplified set of initial conditions with the two-phases training process, resulting in a lower terminal velocity and position errors and a very aggressive control action. In particular as shown

in Fig.5.1a for the 3DOF case, the thrust profile has almost a bang-bang shape, which would be optimal to minimize fuel consumption. A similar profile appeared for the 6DOF case. In the case of realistic conditions, the terminal velocity and position errors were higher, although still compatible with a successful landing. It is interesting to notice that in the 3DOF case the errors are lower, thus there is room for improvement in the policy for the 6DOF environment.

For the 6DOF environment with realistic initial conditions, using a terminal reward that is shaped rather than piece wise significantly improves convergence. Overall the controller developed results in having a trajectory with fuel consumption close to the optimal one. The controller is quite robust to uncertainty in the dynamics of the plant and to external disturbances, hinting that the neural controller has successfully learned to achieve the terminal goal rather than simply associating a certain action to a state in an open-loop fashion.

Quantification of architecture robustness Policy robustification has been implemented during training through *domain randomization*, aiming at training the agent starting from a variety of samples of initial conditions from an *initial conditions space*. The proposed architecture for direct control of the gimbaled thruster and the learned neurocontroller has proven to have a significant level of robustness in the trained scenarios, with a narrow distribution for both terminal position and velocity errors. It is important to highlight however the presence of a low number of outliers: in these episodes the lander reaches ground significantly outside the safe landing bounds. These outliers should be addressed for the policy to be deployable on real hardware.

Reinforcement learning proves to be a promising approach to solve the planetary landing problem in a novel way. It brings an advantageous integrated G&C paradigm, capable of successfully achieving pinpoint landing in a fuel-efficient manner. The obtained policy can be deployed in a straightforward way to COTS microcontroller/embedded computers to perform real-time control of the vehicle.

A key takeaway is that the reward function is critical in obtaining a good solution and is the most difficult part of the algorithm to optimize. Furthermore, hyperparameters must be set appropriately in order to have good convergence behavior and should, when possible, be tweaked beforehand. It would be interesting to analyze whether the optimal hyperparameters found in this work generalize to other vehicle architectures and different dynamics (still using the PPO algorithm).

7.2 Future development directions

Several key areas are open to be explored, and it can be done so by employing the tools developed in this thesis.

Different algorithms In future work it would be interesting to compare the performance and robustness of *model-based* approaches exploiting the intrinsic dynamics of the vehicle, possibly using simplified or learned models and integrating them with *model-free* augmentations, such as with *imagination-augmented* algorithms [30], which would be able to exploit knowledge of a simplified version of the system dynamics to speed up convergence of the policy, retaining a good degree of exploration through the *model-free* pathway.

Another type of algorithm which has proven interesting is *Hindsight Experience Replay* (HER) [47], which avoids the need to heavily shape the reward by being more sample-efficient in simulations.

Training parallelization During training, it could be noticed that the CPU was used only up to about 30% of its maximum load. This is due to the fact that the training algorithm was running on a single one of its multiple cores. A potential future avenue to cut down on the long training times could be to employ parallelization to train faster, by running the rollouts on all the CPU cores. This type of approach is employed in frameworks such as Nvidia's Isaac Gym [48].

Reward functions A crucial role in shaping the reward was the ability to visualize and compare across different experiment runs how adding and removing terms and tweaking its coefficients impact convergence and the resulting trajectories obtained using the optimized policy.

Indeed, further optimization of the reward function or usage of different, possibly learned reward functions (i.e. from 3DOF optimal trajectories computed using non-linear optimizers) could lead to improved fuel efficiency and faster convergence times. In particular for the latter, it would be helpful to have dense reward functions, capable of hint the agent throughout the episode how to achieve a fuel-optimal landing.

Policy robustification The policy proves to be fairly robust to unmodeled disturbances and parametric uncertainties of the model, however the presence of some outliers needs to be addressed, either by further tuning of the reward function or by introducing more uncertainties in the training process, such as action noise or random parameters noise, to robustify the policy. This is critical to achieve a high-reliability implementation and to transition towards real-life testing of such type of controllers.

Explainability An important aspect to be addressed is to understand how the controller selects actions to perform. This property, named in the AI community *explainability*, is critical for aerospace applications and for future certification. There are several ways to do this, such as ablating the input given to the network to isolate the features of the observations that are key for the controller to compute

the action, performing ablation studies on the neural network produced by the Reinforcement Learning algorithm and employing reachability analysis to define a reachable set from the uncertain starting set [49].

Additional control actuators Given the modular nature of the developed Python environment it would be interesting to evaluate the impact of additional actuators (grid fins, RCS control thrusters, etc.) on the convergence behavior of the algorithm and the performance of the obtained trajectory, assessing the trade-off of fuel savings for additional complexity and cost. This effort is ongoing as an implementation of a grid-fin enabled simulator is underway.

Expanded range of initial conditions Development of a single controller for the whole descent profile could be carried out by using techniques such as reinforcement meta-learning, leveraging on the developed work on a single phase of the descent trajectory.

This approach to the development of a guidance and control solution proved to be very promising and could allow higher flexibility for future missions due to the possibility of controlling a wide range of dynamics using several control architectures in a straightforward manner and the capability of defining both high and low-level goals for the agent to achieve.

Bibliography

[1] Deimos Space. `https://elecnor-deimos.com/`,.

[2] Allan R Klumpp. Apollo lunar descent guidance. *Automatica*, 10(2):133–146, 1974.

[3] Christopher D'Souza and Christopher D'Souza. *An optimal guidance law for planetary landing*. 1997.

[4] Behcet Acikmese and Scott R. Ploen. Convex programming approach to powered descent guidance for mars landing. *Journal of Guidance, Control, and Dynamics*, 30(5):1353–1366, 2007.

[5] Lars Blackmore, Behçet Açikmeşe, and Daniel P. Scharf. Minimum-landing-error powered-descent guidance for mars landing using convex optimization. *Journal of Guidance, Control, and Dynamics*, 33(4):1161–1171, 2010.

[6] Behçet Açıkmeşe and Lars Blackmore. Lossless convexification of a class of optimal control problems with non-convex control constraints. *Automatica*, 47(2):341–347, 2011.

[7] Behçet Açıkmeşe, Jordi Casoliva, John Carson, and Lars Blackmore. G-fold: A real-time implementable fuel optimal large divert guidance algorithm for planetary pinpoint landing. *LPI Contributions*, pages 4193–, 06 2012.

[8] Michael Szmuk, Taylor P. Reynolds, and Behçet Açıkmeşe. Successive convexification for real-time six-degree-of-freedom powered descent guidance with state-triggered constraints. *Journal of Guidance, Control, and Dynamics*, 43(8):1399–1413, 2020.

[9] Jacopo Guadagnini, Michèle Lavagna, and Paulo Rosa. Model predictive control for reusable space launcher guidance improvement. *Acta Astronautica*, 193:767–778, 2022.

[10] Marco Sagliano, Taro Tsukamoto, Shinji Ishimoto, Stefano Farì, Etienne Dumont, David Seelbinder, Markus Schlotterer, Ansgar Heidecker, José A. Macés-Hernández, and Svenja Woicke. Robust control for reusable rockets via structured h infinity synthesis. 06 2021.

[11] Pedro V M Simplicio. *Guidance and Control Elements for Improved Access to Space : from Planetary Landers to Reusable Launchers*. PhD thesis, Department of Aerospace Engineering, Bristol Doctoral College, 2019.

[12] Maksim Shirobokov, Sergey Trofimov, and Mikhail Ovchinnikov. Survey of machine learning techniques in spacecraft control design. *Acta Astronautica*, 186:87–97, 2021.

[13] Harry Holt, Roberto Armellin, Nicola Baresi, Yoshi Hashida, Andrea Turconi, Andrea Scorsoglio, and Roberto Furfaro. Optimal q-laws via reinforcement learning with guaranteed stability. *Acta Astronautica*, 187:511–528, 2021.

[14] Roberto Furfaro, Andrea Scorsoglio, Richard Linares, and Mauro Massari. Adaptive generalized zem-zev feedback guidance for planetary landing via a deep reinforcement learning approach. *Acta Astronautica*, 171:156–171, 2020.

[15] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489–501, 2006. Neural Networks.

[16] Yu Zhang, Zheng Fang, and Hongbo Li. Extreme learning machine assisted adaptive control of a quadrotor helicopter. *Mathematical Problems in Engineering*, 2015:905184, May 2015.

[17] Bernd Dachwald. Optimization of very-low-thrust trajectories using evolutionary neurocontrol. *Acta Astronautica*, 57(2):175–185, 2005. Infinite Possibilities Global Realities, Selected Proceedings of the 55th International Astronautical Federation Congress, Vancouver, Canada, 4-8 October 2004.

[18] Lin Cheng, Zhenbo Wang, Fanghua Jiang, and Chengyang Zhou. Real-time optimal control for spacecraft orbit transfer via multiscale deep neural networks. *IEEE Transactions on Aerospace and Electronic Systems*, 55(5):2436–2450, 2019.

[19] Lin Cheng, Zhenbo Wang, and Fanghua Jiang. Real-time control for fuel-optimal moon landing based on an interactive deep reinforcement learning algorithm. *Astrodynamics*, 3:375–386, 07 2019.

[20] Dario Izzo and Ekin Öztürk. Real-time guidance for low-thrust transfers using deep neural networks. *Journal of Guidance, Control, and Dynamics*, 44(2):315–327, 2021.

[21] Maksim Shirobokov and Sergey Trofimov. Adaptive neural formation-keeping control for satellites in a low-earth orbit. *Cosmic Research*, 59:501–516, 11 2021.

[22] K. KrishnaKumar, S. Rickard, and S. Bartholomew. Adaptive neuro-control for spacecraft attitude control. *Neurocomputing*, 9(2):131–148, 1995. Control and Robotics, Part II.

[23] Tim Salzmann, Elia Kaufmann, Jon Arrizabalaga, Marco Pavone, Davide Scaramuzza, and Markus Ryll. Real-time Neural-MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms. *arXiv e-prints*, page arXiv:2203.07747, March 2022.

[24] Brian Gaudet and Roberto Furfaro. Adaptive pinpoint and fuel efficient mars landing using reinforcement learning. *IEEE/CAA Journal of Automatica Sinica*, 1(4):397–411, Oct 2014.

[25] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn, 2016.

[26] Brian Gaudet, Richard Linares, and Roberto Furfaro. Deep Reinforcement Learning for Six Degree-of-Freedom Planetary Powered Descent and Landing. *arXiv e-prints*, page arXiv:1810.08719, October 2018.

[27] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[28] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[29] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. *None*, 2018.

[30] Théophane Weber, Sébastien Racaniere, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017.

[31] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[33] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[34] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395, Beijing. China, 2014. PMLR.

[35] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017.

[36] ISO/TC 20/SC 6 Standard atmosphere. *ISO 2533:1975 Standard Atmosphere*. International Organization for Standardization, Geneva, Switzerland, 1975.

[37] SpaceX. Falcon 9 user guide. Technical report, SpaceX, 04 2020.

[38] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[39] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[40] Declan Murphy. Flightclub, 2022.

[41] Yanning Guo, Matt Hawkins, and Bong Wie. Waypoint-optimized zero-effort-miss/zero-effort-velocity feedback guidance for mars landing. *Journal of Guidance Control and Dynamics*, 36(3):799–809, 2013.

[42] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[43] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[44] C. Bane Sullivan and Alexander A. Kaszynski. Pyvista: 3d plotting and mesh analysis through a streamlined interface for the visualization toolkit (vtk). *Journal of Open Source Software*, 4(37):1450, 2019.

[45] Zachary Peterson, Shannon Eilers, and Stephen Whitmore. *Closed-Loop Thrust and Pressure Profile Throttling of a Nitrous-Oxide HTPB Hybrid Rocket Motor*.

[46] Newlands Rick, Heywood Martin, and Lee Andy. Rocket vehicle loads and airframe design. technical paper, 2016.

[47] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[48] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance gpu-based physics simulation for robot learning, 2021.

[49] Haimin Hu, Mahyar Fazlyab, Manfred Morari, and George J. Pappas. Reach-sdp: Reachability analysis of closed-loop systems with neural network controllers via semidefinite programming. 2020.