Spring 2020

# Landing Throttleable Hybrid Rockets with Hierarchical Reinforcement Learning in a Simulated Environment

Francesco Alessandro Stefano Mikulis-Borsoi
*University of New Hampshire*

# UNIVERSITY OF NEW HAMPSHIRE

**Bachelor of Science with Honors Degree in Computer Science**


**Honors Thesis**

# Landing Throttleable Hybrid Rockets with Hierarchical Reinforcement Learning in a Simulated Environment

Repository: https://github.com/francescomikulis/rocketlander

**Supervisors**

**DR. MAREK PETRIK**

**Candidate**

**FRANCESCO ALESSANDRO STEFANO MIKULIS-BORSOI**

**May 2020**

# Abstract

In this paper, I develop a hierarchical MDP structure for completing the task of vertical rocket landing. I start by covering the background of this problem, and formally defining its constraints. In order to reduce mistakes while formulating different MDPs, I define and develop the criteria for a standardized MDP definition format. I then decompose the problem into several sub-problems of vertical landing, namely velocity control and vertical stability control. By exploiting MDP coupling and symmetrical properties, I am able to significantly reduce the size of the state space compared to a unified MDP formulation. This paper contains two major contributions: 1) the development of a standardized MDP definition framework and 2) a hierarchical MDP structure that is able to successfully land the rocket within the goal bounds more than 95% of the time. I validate this approach by comparing its performance to a baseline RRT search, underlining the advantages of rapid-decision making compared to online planning.

# Acknowledgements

Firstly, I would like to thank my thesis advisor Dr. Marek Petrik of the Computer Science department at the University of New Hampshire, who inspired me to pursue my interests in Machine Learning. By pushing me to take his most advanced courses, Dr. Petrik disseminated seeds of curiosity throughout my studies, challenging me to evolve my approach and thinking of Machine Learning.
I also want to thank the Computer Science PhD candidate Paul Gesel, who worked with me for extensive hours and many late nights to review my hierarchical MDP structure. Not only did he oversee the implementation of difficult physics concepts such as thrust vectoring, but he also reviewed this thesis and encouraged its expansion.
I cannot forget to acknowledge Dr. Radim Bartos, the Chair of the Computer Science Department at the University of New Hampshire. His support and confidence in my abilities empowered me to join this top-of-the-line research university.
I express my most profound gratitude to my mother, Elizabeth Anne Mikulis, for providing me with unparalleled support and encouragement throughout the entirety of my academic career. Her demanding expectations empowered my curiosity and propelled me in the direction for success.
Finally, I am grateful to my girlfriend, Giorgia, and to my family and friends for their unbounded support, patience and understanding of my academic commitments and workaholic tendencies. Their emotional support, encouragement, especially in times of need assisted in propelling me forward, even at times when I doubted my own capabilities.
This accomplishment would not have been possible if it wasn't for the support of my UNH professors, family and friends.

Thank you,

Francesco Alessandro Stefano Mikulis-Borsoi

# Statement of Collaboration

The results presented in this thesis were developed over a 9-month period (from September 2019 to May 2020). Specifically, I obtained many of these results during group-projects in graduate level courses. The following description contains: the courses in which I expanded this thesis, the names of the students I worked with, and their specific contribution to this work.

- CS 950 (Advanced Machine Learning) - Fall 2019:
  - Paul Gesel
    - primarily implemented thrust vectoring in OpenRocket
    - proposed Blender for visualization and created the virtual rocket model
    - discussed different MDP implementations
    - supported initial developments of the code-base via peer-programming
  - Kristian Comer
    - introduced me to the open-source rocket simulator OpenRocket
- CS 853 (Introduction to Artificial Intelligence) - Spring 2020:
  - Paul Gesel
    - proposed the decomposition of the main controller into several task-specific controllers based on a control theory approach
    - developed a mathematical model of the rocket's dynamics
    - proposed and mainly implemented the RRT search baseline

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

| | |
|---|---|
| **2D** | 2-dimensional |
| **3D** | 3-dimensional |
| **AI** | Artificial Intelligence |
| **GUI** | Graphical User Interface |
| **MC** | Monte Carlo (the reinforcement learning algorithm) |
| **MDP** | Markov Decision Process |
| **MPC** | Model Predictive Control |
| **ML** | Machine Learning |
| **OR** | OpenRocket (the software application) |
| **RL** | Reinforcement Learning |
| **RRT** | Rapidly exploring Random Rrees |
| **SDF** | Standard Definition Framework |
| **SDF-MDP** | Standard Definition Framework for Markov Decision Processes |
| **TCP** | Transmission Control Protocol |
| **TD0** | Temporal Difference 0 (the reinforcement learning algorithm) |
| **UI** | User Interface |

# Symbols

I will use the symbolic notation below throughout the paper. Notably, elements of vectors are indexed with a single subscript, where the subscript of x, y, or z indicate the first, second, or third component of a vector, respectively. For example, $S_x$ indicates the first component of the vector $S$. Some symbols are written with superscript notation. This is only for notation purposes and does not represent a mathematical operation. For example, the the superscript $r$ in $R^r$ designates a that $R$ is the rotation matrix from the rocket's coordinates to the global coordinates. Furthermore, the dot ($\dot{x}$) notation represents a time derivative of $x$.

| | |
|---|---|
| $t$ | time |
| $x$ | position along x axis in global coordinates |
| $y$ | position along y axis in global coordinates |
| $z$ | position along z axis in global coordinates |
| $\dot{x}$ | velocity along x axis in global coordinates |
| $\dot{y}$ | velocity along y axis in global coordinates |
| $\dot{z}$ | velocity along z axis in global coordinates |
| $\omega$ | angular velocity in global coordinates |
| $u$ | control vector |
| $\theta_x$ | gimbal angle about its x axis |
| $\theta_y$ | gimbal angle about its y axis |
| $\gamma_x$ | rocket angle about its x axis |
| $\gamma_y$ | rocket angle about its y axis |
| $T(t)$ | non-throttled rocket thrust force |
| $F$ | force applied to the rocket in global coordinates |
| $\tau$ | torque applied to the rocket in global coordinates |
| $g$ | force of gravity on rocket |
| $f(x, w)$ | unknown forces and torques |
| $m$ | mass of the rocket |
| $cg$ | center of gravity |
| $cm$ | vector from the cg to the gimbal in rocket coordinates |
| $l$ | vector from the cg to the gimbal in global coordinates |
| $R^r$ | rotation matrix from rocket coordinates to global coordinates |
| $d$ | gimbal direction in rocket coordinates |
| $D$ | rocket axial direction in global coordinates |
| $a$ | action |
| $A$ | set of available actions |
| $S$ | state vector |
| $r(S)$ | reward function evaluated on state vector S |
| $r_t(S)$ | terminal reward function evaluated on state vector S |

# Chapter 1

# Introduction

The objective of this paper is to develop a controller that can land a rocket vertically in a simulated 3D environment. Self-landing rockets have been theorized over the past century, but until 2015, no one had successfully completed vertical landing with thrust vectoring in a real rocket. The controller used to solve this type of problem is required to reach predefined goal conditions from a variety of starting states. A popular approach for vertical rocket landing includes MPC, such as that seen in R. Ferrante [1], but in most cases, the problem is solved in a 2D environment. I will solve this problem with Reinforcement Learning (RL) under a hierarchical Markov Decision Process (MDP) framework in 3D.

## 1.1 Complexity of Autonomous Rocket Landing

Developing a system to autonomously complete the task of rocket-landing is a complex problem, containing the following characteristics:

- Continuous state definition: Each of the variables in this problem are defined on a continuous interval of the real numbers.
- Partially observable state-space: I later develop a simplified formulation of this problem, which ignores certain rocket dynamics and less important physical information in order to reduce the number of variables, effectively causing the state-space to be partially observable.
- Non-linear dynamics: The thrusting forces as well as the aerodynamic forces cause the rocket to be subject to non-linear dynamics during the landing task.
- Requirement of real-time control: In order to ensure high success-rate while landing the rocket, one must guarantee that the controlling actions be real-time.
- Financially prohibitive verification: Building a rocket requires significant financial commitment and years of testing, thus real trials can only be attempted once a formal problem solution obtains a near perfect success-rate.

## 1.2 A Brief History

The rocketry domain has expanded since the early ‘40s, in which Dr. Wernher Von Braun researched the use of high powered rockets, a technology that was later used by the Germans in the V-2 ballistic missiles on London in WWII [2]. As discussed by the National Academy of Engineering [3], throughout the last 60 years different autonomous spacecrafts have enabled humans to return from space, land rovers on the surface of Mars [4], navigate to Titan [5] as well as land on different asteroids [6].

The rocketry community has focused its efforts on launching rockets, and for a long time discarded the concept of landing the rockets using its thrusters, resorting instead to a multi-parachute approach. Self-landing rockets have been theorized over the past century but until 2014 no one had ever managed to bring a rocket into outer space and successfully complete a stable re-entry with a thrust-landing. The most common approach of controlling a rocket during the re-entry phase is Model Predictive Control (MPC), a technique that requires to forward simulate the dynamics of the rocket. The issue with this approach is that it requires an extremely precise model of the rocket in question and it has large computational requirements.

## 1.3 Commercial Applications

### 1.3.1 SpaceX and Reusable Rockets

Since 2014, SpaceX has continuously developed autonomous self-landing commercial rockets. The efforts towards developing a self-landing rocket are driven by multiple commercial factors: losing or damaging parts of a commercial space-bound rocket costs millions of dollars, and re-using the same rocket is infeasible with the traditional recovery system of multiple parachutes. In fact, throughout the past 6 years SpaceX has been able to maintain financial stability by obtaining contracts with the US government to deliver cargo supplies to the International Space Station (ISS) [7]. The second major reason for developing self-landing rockets is for the ability to land on Mars. By mastering thruster landings solely based on the Earth's gravitational pull, such techniques can be re-applied for a landing on Mars, where the atmosphere is much less dense, and a parachute would be unable to slow the landing of a large rocket.

### 1.3.2 Implications of Solving This Problem

As discussed earlier, this problem is defined in a continuous 3-dimensional state-space and has high fidelity with the real world. Many problems with the same complexity characteristics have been active fields of academic research, and unfortunately only some of the problems with the greatest economic potential have been approximately solved. The objective of this paper is to demonstrate the effectiveness of hierarchical MDPs in these types of 3-dimensional complex decision making problems. An example that relates to this paper is discussed by P. Lu in [8], in which a non-linear predictive controller is developed for an application in missile autopilot design.

## 1.4 Related Work

Approaches involving autonomous rocket-landing have been overwhelmingly developed under the influence of control theory, which requires solving approximated physical dynamics of the real world at each decision step of the system. This approach is known as the receding horizon and is achieved through simulated steps. Long term predictions are infeasible because incremental errors accumulate during each simulation step, thus decisions must be made with a limited finite horizon. SpaceX uses a control theory based approach, with an on-board computer leveraging a receding horizon to predict its state by generating an open-loop state and control trajectories. Other individuals have analyzed the performance comparison between MPC and RL methods, such as R. Ferrante [1], but often the analysis is approached in a 2-dimensional environment, with the aid of lateral thrusters.

## 1.5 The Case for MDPs

### 1.5.1 A Philosophical Perspective

The policy learned with an MDP formulation is reflexive. That is, the policy directly maps state to action without online planning. Hence, an agent acting in accordance to a learned policy is essentially a reflexive agent. I pose the question of whether or not a reflex agent exhibits intelligence. One might ask, "what is intelligence?" According to the Goertzel, the capability of an agent to achieve complicated goals in dynamic environments is a suitable interpretation of intelligence [9]. I take the perspective that intelligence is neither reflex nor planning, but a combination of both. The study of the structures of consciousness under the first-person point of view - commonly known as phenomenology - was explored by Sutton and many other researchers, whom argue that acting appropriately in the heat of the moment is still a form of intelligence [10].

This is further reinforced by Krakauer, who argues that human tasks at many levels of expertise are composed of intelligent reflexes and deliberative decisions [11].

A reflex agent on its own may not be intelligent, but an agent that intentionally develops its reflexes for skills that it will need is brilliant. Arguably, developing and remembering a reflex for a repeated task is a critical component of intelligence. My proposed hierarchical MDP structure aims to capture this phenomenon; the role of reflex in intelligent systems. I implement low level controllers (or MDPs), which resemble the reflexes of an intelligent system. Under the hierarchical MDP structure, both deliberate decision making and intelligent reflexes can be modelled.

### 1.5.2 A Practical Perspective

A practical issue of online planning is encountered when edge evaluations or node expansions are computationally expensive. Search methods, such as Lazy Weighted A* [12], aim to alleviate this problem by postponing expensive computations until they are absolutely necessary. Even a computationally aware planner must ultimately perform a non-optimal number of edge evaluations if the optimal heuristic is unknown. Complex physics problems frequently require massive computational power. For example, simulating a rocket engine in an optimized computational fluid dynamics library can require several days of CPU time [13]. This effectively eliminates online search as an option in a real-time setting. Other practical benefits of direct state-action mapping include ease of implementation on low level hardware and low computational cost.

# Chapter 2

# Background

In the following sections sections, I will derive relevant theoretical models for thrust vectoring rocket dynamics. This is followed by a high level overview of MDPs, coupled MDPs, symmetrical MDP properties, hierarchical MDPs, and finally RRT.

## 2.1 Rocket Model

In this section, I will derive the dynamics of a throttleable thrust vectoring rocket. Notably, the implementation will rely on an open-source software to simulate the rocket's dynamics. The model derived in this section provides insight on vertical rocket landing, which will help us create reasonable state definitions in the later sections.

The model has 5 actuators: one to throttle the main thruster, two to control the gimbal angles of the thruster, and two more to set the force of the lateral thrusters. The unthrottled force of the main thruster is denoted $T(t)$. Notably, the rocket is under-actuated and cannot move the gimbal while using lateral thrusters. More specifically, the pair of actions $u_2$ and $u_5$ and the pair of actions $u_3$ and $u_4$ cannot be changed at a single instant. Fig. 2.1 illustrates a free body diagram with all forces and torques modeled.

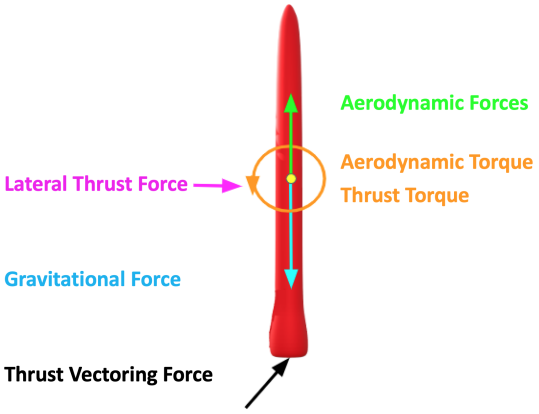



**Figure 2.1:** Rocket free body diagram.

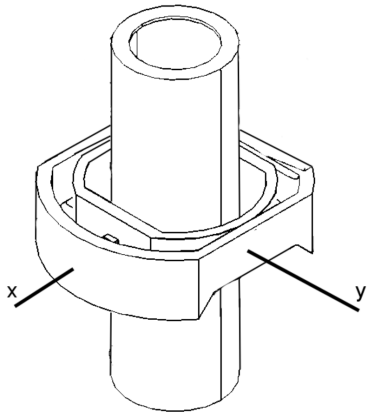



**Figure 2.2:** Schematic of the gimbal design.

The black arrow indicates the thrust vectoring force $F$. Since it is applied at the bottom of the rocket, causing a torque $\tau$ (shown in orange). The purple arrow indicates the lateral thruster force $F^l$. This is applied at the center mass of the rocket and perpendicular to the rocket's $z$ axis. The remaining arrows represent gravity $g$ and unknown aerodynamic forces. Since the aerodynamic forces depend on unattainable sensor measurements, such as temperature and wind, those forces are denoted as forces and torques as $f(x, w)$, where $w$ is an unobserved variable. The rocket's state vector is defined in the following way,

$$S = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \gamma_x, \gamma_y, \omega_x, \omega_y]^T \in \mathbb{R}^{10}$$

where $x$, $y$, and $z$ represent the rocket's position, $\dot{x}$, $\dot{y}$, and $\dot{z}$ represent the rocket's velocity, $\gamma_x$ and $\gamma_y$ indicate the rocket's pitch and yaw, and $\omega_x$ and $\omega_y$ are the rocket's angular velocities. The dot notation indicates the time derivatives of each variable. With this definition, the rocket's orientation is expressed as follows.

$$R^r = \begin{bmatrix} \cos(\gamma_x) & 0 & \sin(\gamma_x) \\ \sin(\gamma_y)\sin(\gamma_x) & \cos(\gamma_y) & -\sin(\gamma_y)\cos(\gamma_x) \\ -\cos(\gamma_y)\sin(\gamma_x) & \sin(\gamma_y) & \cos(\gamma_y)\cos(\gamma_x) \end{bmatrix} \tag{2.1}$$

I decided not to model roll for two reasons:

1. None of the modeled forces create a torque about the rocket's axial direction.
2. If the rocket does roll, none of the controls can stop the motion.

The five available controls are defined as follows.

$$u = [u_1, u_2, u_3, u_4, u_5]^T \in \mathbb{R}^5$$

where $u_1$ is the throttle multiplier of the main thrust, $u_2$ and $u_3$ are the gimbal angles, and $u_4$ and $u_5$ are the lateral trust forces in the $x$ and $y$ directions, respectively. Finally, the vector from the rocket's gimbal to its center of mass in global coordinates is expressed as $l$ and force of gravity on rocket $g$.

$$l = R^r \begin{bmatrix} cm_x \\ cm_y \\ cm_z \end{bmatrix} \tag{2.2}$$

$$g = \begin{bmatrix} 0 \\ 0 \\ -9.81m \end{bmatrix} \tag{2.3}$$

### 2.1.1 Thrust Vectoring Dynamics

The gimbal is composed of two revolute joints; one with its axis of rotation in x and one in y, as shown in Fig. 2.2. These angles are denoted as $\theta_x$ and $\theta_y$, respectively. The kinematics of this design are expressed below, where $d$ represents the direction of the gimbal in rocket coordinates.

$$d = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_y) & -\sin(\theta_y) \\ 0 & \sin(\theta_y) & \cos(\theta_y) \end{bmatrix} \begin{bmatrix} \cos(\theta_x) & 0 & \sin(\theta_x) \\ 0 & 1 & 0 \\ -\sin(\theta_x) & 0 & \cos(\theta_x) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

$$d = \begin{bmatrix} -\sin(\theta_x) \\ \sin(\theta_y)\cos(\theta_x) \\ -\cos(\theta_y)\cos(\theta_x) \end{bmatrix} \tag{2.4}$$

Since the gimbal direction is a unit vector, the third component can be written as a function of the first two components.

$$d = \begin{bmatrix} -\sin(\theta_x) \\ \sin(\theta_y)\cos(\theta_x) \\ -\sqrt{1 - \sin^2(\theta_x) - \sin^2(\theta_y)\cos^2(\theta_x)} \end{bmatrix} \tag{2.5}$$

In global coordinates, the force due to the gimbal is expressed as

$$F^g = -R^r dT(t) u_1 \tag{2.6}$$

and the force due to the lateral thruster is shown below.

$$F^l = R^r \begin{bmatrix} u_4 \\ u_5 \\ 0 \end{bmatrix} \tag{2.7}$$

The lateral thrusters generate no torque on the rocket since they are positioned at the center of mass. The torque can be written as the cross product between $l$ and $F^g$.

$$\tau = l \times F^g \tag{2.8}$$

The total sum of forces and torques on the rocket are expressed below as a function of the gimbal direction, the main thrust, and the unknown forces.

$$\begin{bmatrix} F_x \\ F_y \\ F_z \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} F^g + F^l + g \\ l \times F^g \end{bmatrix} + f(x, w) \tag{2.9}$$

The cross product is a linear operator of the following form.

$$l\times = \begin{bmatrix} 0 & -l_3 & l_2 \\ l_3 & 0 & -l_1 \\ -l_2 & l_1 & 0 \end{bmatrix} \tag{2.10}$$

This highlights the fact that the control of the torque is linearly dependent on the control of the force for a given rocket orientation regardless of control input $u$.

$$\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} 0 & -l_3 & l_2 \\ l_3 & 0 & -l_1 \\ -l_2 & l_1 & 0 \end{bmatrix} \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} \tag{2.11}$$

Notably, at most three out of the five derivative state variables can be independently manipulated, emphasizing the difficulty of this problem.

## 2.2 MDP

An MDP is used to model decision making in discrete, stochastic, sequential environments. Under this framework, an agent observes a state $s_t \in S$ at each time step $t$. The agent must then take an action $a \in A$, while trying to maximize the total expected future reward. This process repeats from $t = 0$ to $t = t_f$, where $t_f$ is the terminal time. The state trajectory is denoted as an episode. The problem is mainly concerned with finite horizon MDPs, which often include a terminal reward function $r_t(S)$. This reward is received only once, when the episode is terminated. The solution to an MDP is a policy $\pi : S \to A$ that maps states to actions, which maximize the total expected reward. Every policy has a concept of a value function $V(S)$ as seen in equation (2.12). The value function indicates the total expected future reward from a given state if the policy $\pi$ is followed. The optimal policy can be found by solving for the optimal value function $V^*(S)$ with value iteration [14].

$$V^*(s) = \max_a \sum_{s'} P(s'|a, s) \left[ r(s, a, s') + \gamma V^*(s') \right] \tag{2.12}$$

The Q-learning algorithm in equation (2.13) follows a format similar to expression (2.12), except that the transition probabilities $P$ are not known [14].

$$Q(s, a) = Q(s, a) + \alpha \left[ r(s) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{2.13}$$

This is known as model free and is advantageous since an explicit model of the process is not required. In this setting, an action value function is used to learn the expected reward of taking any action from any state. With this peculiar formulation that avoids $P$, one realizes in Q-learning the number state action tuples grows exponentially with the number of states and actions. Yet this issue can also be overcome, by leveraging the Q-learning algorithm and using value function approximation.

### 2.2.1 Coupled MDPs

One general method for tackling large MDPs is decomposition [15]. This can be applied when an MDP is specified by a set of "pseudo-independent" sub-processes [16]. Tightly-coupled MDPs are a set of MDPs where the action of one MDP influences the state and available actions of other MDPs. Traditionally, this solution framework is effective when there is a "main" controlling action that directly influences other available actions. For the problem setup, the "main" control action is the selection of the thruster power - a construct parallel to the available power of the rocket in a given state.

### 2.2.2 Symmetrical Properties in MDPs

The concept of exploiting symmetry in MDPs has been analyzed in [17] and [18]. The results of these papers confirm that policies can be formulated as functionally homogeneous. The proof of the optimality of the resulting MDP requires the following background definitions.

**Definition 1** *A **Multiagent Markov Decision Process** $(MMDP)$ with n agents is an MDP $(S, A, T, R)$ where $S \subseteq (S_{agent})^n$ for some set $S_{agent}$ and $A = (A_{agent})^n$ for some set $A_{agent}$, where $S_{agent}$ is considered the state space of a single agent, and $A_{agent}$ is considered the set of actions of a single agent.*

**Definition 2** *A **permutation** of a state is a permutation of the elements. If P is a permutation of a set of vectors $V^n$, then there exists a bijection g: $\{1 \ldots n\} \rightarrow \{1 \ldots n\}$, such that $\forall \vec{v} \in V^n$, $\forall i P(\vec{v})_i = v_{g(i)}$. The inverse of a permutation P is the permutation $P^{-1}$ such that for all $\vec{v} \in V^n$, $\vec{v} = P^{-1}(P(\vec{v}))$. The inverse of the inverse of P is P.*

**Definition 3** *An **equivalence homogeneity** for an MMDP is an ordered pair $(H_S, H_A)$ where $H_S$ and $H_A$ are defined as follows: 1. $(s, s') \in H_S$ if and only if there exists a permutation P such that $s' = P(s)$. 2. $((s, a), (s', a')) \in H_A$ if and only if there exists a permutation P such that $s' = P(s)$ and $a' = P(a)$.*

**Definition 4** *An MMDP is **functionally homogeneous** if for all permutations P, all states s, s' $\in$ S, and all actions $a \in A$, $T(P(s), P(a), P(s')) = T(s, a, s')$ and $R(P(s), P(a)) = R(s, a)$*

**Theorem 1** *A functionally homogeneous MMDP with agents in distinct states possesses a homogeneous optimal policy.*

This means that if a multiagent MDP forms an equivalent homogeneity, then there exists an optimal policy which is symmetric with respect to their equivalence homogeneity.
For example, let us consider the scenario in which an MDP learns the optimal policy for playing the game tic-tac-toe on a board of size 3x3, where each symbol occupies a single square of size 1x1. Now, let us execute this same policy on a board of size 9x9: the MDP's policy will still be optimal if each symbol occupies a size of 3x3 and each action's coordinate is multiplied by 3 in the x and y axis of the grid.
Additionally, the approach of exploiting symmetry is expected to speed up the convergence to the optimal value function, because the agent learns the strategy for a "smaller" MDP. In the rocket landing problem, I exploit the symmetry of the system, namely in controlling the gimbal and the lateral thrusters.

### 2.2.3 Hierarchical MDPs

The motivation for a hierarchical MDP is mitigating the curse of dimensionality by solving the problem in an abstract space [14]. Since memory and computational requirements grow exponentially with the number of states and actions, reducing the state space through a hierarchical structure can lead to tractability. The concept of options is a poplar hierarchical scheme proposed

by Sutton [19]. In this setting, an MDP is allowed to chose an option MDP as one of its actions. The chosen MDP executes until a termination criteria is met, upon which control is returned to the original MDP. Since the selected MDP executes for a variable amount of time, this results in a semi-MDP.

In hierarchical MDPs, the structure plays a critical role in their performance. Poorly chosen option MDPs will not be beneficial. Some methods have been proposed to automatically build a hierarchical MDP, such as MAXQ [20]. However, the policy learned from MAXQ is not globally optimal, rather it is recursively optimal. Automatic generation of hierarchical MDPs is a difficult task and an ongoing area of research. For the approach, I will inject domain knowledge into the structure by hand crafting relevant sub-MDPs.

## 2.3 RRT - A Search Approach Baseline

Rapidly-exploring random trees (RRT) have shown great success in solving non-convex high-dimensional spaces by building a space-filling tree and were first proposed by Lavalle [21]. Some high dimensional planing domains have seen great success with RRT, such as robotic manipulation [22]. In particular, an RRT-based implementation enabled a dual arm robot with 14 degrees of freedom to perform assembly and disassembly tasks in a industrial scenario. This exemplifies the effectiveness of RRT-based search for high dimensional continuous problems. RRT will be used as a baseline search approach for the problem.

# Chapter 3

# Formal Problem

I formulate vertical rocket landing with several different MDPs, including a hierarchical MDP structure, given the controls and sensor configuration from Tab. 3.1 and Tab. 3.2. The final policy must reach any of the goal states defined in Tab. 3.3 from any initial state defined in Tab. 3.4 with at least a 95 percent success rate. Success is defined as controlling the rocket by using any sequence of available actions which lead the rocket from any initial state to any of the goal states within a 7 second time limit.

| control | min | max | units |
|---|---|---|---|
| thrust | 0.0 | 200 | N |
| gimbal angles (X, Y) | -5 | 5 | ° |
| lateral thrust (X, Y) | -20 | 20 | N |

**Table 3.1:** Available controls with ranges.

| sensor | precision | units |
|---|---|---|
| x, y, z | 0.5 | m |
| $\dot{x}, \dot{y}, \dot{z}$ | 1 | ° |
| $\omega_x, \omega_y$ | 2 | °/s |

**Table 3.2:** Available sensors and their precisions (inertial measurement unit).

| state | min | max | units |
|---|---|---|---|
| z | 0.0 | 0.75 | m |
| $\gamma_x, \gamma_y$ | -4 | 4 | ° |
| $\dot{z}$ | 0.0 | 0.75 | m/s |
| $\dot{x}, \dot{y}$ | -0.75 | -0.75 | m/s |

**Table 3.3:** Ranges of the goal states.

| state | min | max | units |
|---|---|---|---|
| z | 28 | 32 | m |
| $\gamma_x, \gamma_y$ | -20 | 20 | ° |
| $\omega_x, \omega_y$ | -26 | 26 | °/s |
| $\dot{z}$ | -10 | -8 | m/s |
| $\dot{x}, \dot{y}$ | -6 | 6 | m/s |

**Table 3.4:** Ranges of varying initial states.

Given the continuous nature of this problem, I confined the state of the rocket to be within the ranges specified in Tab. 3.5. Additionally, the actions of the rocket where selected from the ranges specified in Tab. 3.6. Perhaps, violating any of the state boundaries would be sufficient to terminate the simulation and invoke a strong penalty to the MDP. I opted instead to let the MDP explore states outside of these boundaries until the simulation time reached 7 seconds. At this point, if the rocket was over 0.5 m in $z$, the simulation was forcefully terminated with a large

| state | min | max | units |
|---|---|---|---|
| x, y | -20 | 20 | m |
| z | 0 | 32 | m |
| $\dot{x}, \dot{y}$ | -20 | 20 | m/s |
| $\dot{z}$ | -10 | 2 | m/s |
| $\gamma_x, \gamma_y$ | -45 | 45 | ° |
| $\omega_x, \omega_y$ | -45 | 45 | °/s |

**Table 3.5:** Selected state bounds.

| control | min | max | units |
|---|---|---|---|
| thrust | 20 | 180 | N |
| gimbal angles (X, Y) | -3 | 3 | ° |
| lateral thrust (X, Y) | -18 | 18 | N |

**Table 3.6:** Selected action bounds.

penalty. The 7 second simulation boundary was chosen because at this point the rocket's thruster burns out, causing it to lose control.

## 3.1 The OpenRocket Simulator

In order to ensure accurate simulations, I opted for the most realistic open-source rocket simulator available today. I selected OpenRocket [23], a model-rocket simulator actively used by thousands of members of the rocketry community. A model-rocket simulator is precise enough for the problem because I am operating at low altitude, where wind and other variables have low variation and are easier to model.

### 3.1.1 Positive Aspects

This application has over 273 stars on GitHub and over 30 contributors, increasing my confidence on its reliability. Its simulator leverages the 4th order method of Runge–Kutta (RK4), which uses temporal discretization to approximate numerical solutions of ordinary differential equations [24]. Additionally, OpenRocket supports a framework with simulation listeners, drastically reducing changes required to the source code of the simulator. The application is implemented to be multi-processed and multi-threaded, by running multiple simulation workers and drawing CPU time from a pool of available threads. Thus, I implemented thread-safe code to allow for parallelizable training sessions.

### 3.1.2 Negative Aspects

OpenRocket did not support thrust vectoring - directing the thrust with a gimbal - and did not support forces from lateral thrusters - perpendicular to the rocket, located at its center of mass. Hence, I added these dynamics by implementing equation (2.9).

# Chapter 4

# A Standardized MDP Definition Format

In this section, I discuss the criteria required for a dynamic MDP definition framework. I start by analyzing the critical requirements of such a framework, discuss the implementation, and explain my reasoning for this approach.

## 4.1 Default State and Action Fields

OpenRocket steps through a simulation by updating a SimulationStatus object. It contains many fields related to the rocket's state, a subset of which are sufficient to define my problem. The inherited **state** fields from the SimulationStatus follow the format "fieldX", "fieldY" and "fieldZ". These state fields are positionX, velocityX, angleX, angleVelocityX, positionY, etc. The **action** fields are: thrust, gimbalX, gimbalY, lateralThrustX, lateralThrustY.

## 4.2 Framework Implementation Requirements

A dynamic and flexible MDP definition framework requires the following criteria, supporting:

- Flexible MDP definition creation in a standardized format.
- MDP definitions in a human-readable and editable format.
- The description of state and action variables.
- A saveable definition file format.
- Arbitrary expression definitions, for reward specification and custom fields (e.g. log scale).
- Parameter specification for different RL methods (discount, learning rate and exploration rate).
- Single MDP implementations.
- Coupled MDP implementations.
- Axial-symmetry of specific fields, to enable the re-use a policy around a axially-symmetric transformation.
- Hierarchical rule-based MDP selection.
- Hierarchical MDP structure where an MDP selects another MDP as an action.

The following criteria are critical, but can rely on additional software:

- Plots for definition boundaries and discretizations.
- 3D visualization for verification of results.

## 4.3 Implementation and Specification

The following paragraphs define the specification format, and a full example of the standardized format is shown below.

```
{
  "name": "hierarchical_selector",
  "methodName": "TD0",
  "priority": 1,
  "reward": "Add(Add(-Pow(Todeg(angle),2.0),0.3),
            -Mult(Abs(log2Velocity),100))",
  "terminalReward": "0",
  "discount": 0.999,
  "stepDiscount": 0.9,
  "alpha": 0.01,
  "exploration": 0.01,
  "symmetryAxes": [
    "angle",
    "angleVelocity",
    "velocity"
  ],
  "passDownSymmetryAxis": true,
  "stateDefinition": {
    "log8Angle": [-0.15, 0.15, 0.03],
    "log2Velocity": [-2.5, 2.5, 0.25]
  },
  "actionDefinition": {
    "selectorMDP": [0.0, 1.0, 1.0]
  },
  "childrenMDPOptions": {
    "selectorMDP": [
      "hierarchical_stabilizer",
      "hierarchical_damper"
    ]
  },
  "expressions": {
    "log8Angle": "Mult(Signum(angle),Log8(Add(Abs(angle),1)))",
    "log2PositionZ": "Log2(Add(positionZ,1))",
    "log2Velocity": "Mult(Signum(velocity),Log2(Add(Abs(velocity),1)))"
  },
  "successConditions": {
    "velocity": [-0.75, 0.75],
    "angle": [-0.0698, 0.0698]
  }
}
```

### 4.3.1 Core Definition

User-specified MDP definitions are implemented as a Java class, which can be entirely populated by a JSON string. By selecting JSON as my specification format, I am able to allow for

MDP definitions to be directly edited and modified without the aid of additional software. An MDP definition has the following required fields: "name", "methodName", "discount", "reward", "stateDefinition" and "actionDefinition". The definition also relies on the fields: "exploration", "alpha" (learning rate), "stepDiscount", "terminalReward" and finally "successConditions" which follows the format "successStateField": [min, max], inclusive.

### 4.3.2 Custom Expressions

User-defined expressions were implemented with recursive-descent parsing, and can be evaluated on a state or action, allowing for a complete customization of the reward and terminal reward functions. Additionally, the values of state variables can be assigned directly from custom expressions, enabling definitions to use non-linear scales. The syntax for mathematical functions must be specified starting with an uppercase character, constants must be entirely uppercase, and finally variables only require the first character to be lowercase. Common mathematical functions were implemented, such as Add, Sum, Sub, Mult and Log. Additionally, basic boolean logic was also implemented (And, Or, Not), where 0 is treated as false and any other value is treated as true.

### 4.3.3 State and Action Descriptions

State and action boundaries and discretizations follow the format of [min, max, increment]. By using this format, I am able to force the state values within the boundaries defined by the specification. If a value is smaller than the min, then it is set to the min; the same construct is used for the max. When an MDP is hierarchical and selects children MDPs, it must specify the field "childrenMDPOptions". This contains a map from the available actions to the available MDPs for that action. If an action specifies the selection of an MDP, the action field name must contain the string "MDP".

### 4.3.4 Symmetry Definitions

The selection of MDPs that use axial symmetry for their variable assignment require the final character of the action field to contain the expected symmetry axis, X or Y. For example, the action controlMDPX selectes an MDP from the "controlMDPX" field of "childrenMDPOptions", and assigns X to the value of that state's symmetry and action's symmetry. The axis string value will be later appended to a given symmetrical field's name for assignment purposes. Axial symmetry of either the X or Y axis can be specified in the "symmetryAxes" field, in which all fields are assigned the value from that component with symmetry defined based on the symmetry of that instance of the state or action. An exemplary use of this field is the use of angle as a state variable, and the adding "angle" in the "symmetryAxes". At runtime, if this MDP is selected for symmetry with the value of the X axis, then the "angle" field will be assigned the value of angleX.

### 4.3.5 Hierarchical MDP Selection Expressions

By default, an MDP that selects MDPs will be hierarchical and will select an available MDP for a given action. It is also possible to use a rule-base selection, by using the field "MDPSelectionExpressions". This definition field is in the format of a switch-case statement with breaks, in which the first true expression will select that MDP, and the default selection is the last MDP.

## 4.4 3D Visualization and Plotting

Given that I am using OpenRocket as the simulator environment, I opted to integrate visualization and plotting directly within their framework. Notably, the 3D visualization listener works independently of the MDP framework, by obtaining all state variables from the simulation. If a

listener extends the "AbstractSimulationListenerSupportsVisualize3DListener", it can define the thrust, gimbal and lateral thrust controls to be used during the visualization. I also extended OpenRocket's built-in plotting capabilities by adding additional state variables to the list of supported plots. In the following paragraphs, I explain how I developed 3D visualization and added custom plotting capabilities to OpenRocket.

### 4.4.1 3D Visualization

I created a simulation listener for 3D visualization, which subscribes to updates of the simulation status at each step of the simulation. This listener was implemented as an interface, allowing for configurable visualization preferences. For the 3D visualization, I opted for an integration with the open-source Blender application, which supports python scripting. I developed a Python script that runs a TCP server on the local IP address with port 8080, creating flexibility by allowing the visualization to run on a different machine. Fig. 4.1 contains a screen capture of the 3D visualization with Blender. The ability to visualize the behavior of an MDP real-time is critically important to verify the correctness of the solution. Additionally, this tool is helpful to analyze unexpected behaviors in an MDP's policy.

**Figure 4.1:** 3D visualization with Blender.

### 4.4.2 Plotting

I implemented custom plotting capabilities for variables defined within the state and action of an MDP definition. An important thing to note is that MDP definition variables are forced within their definition boundaries. Notably, if a custom expression is defined for a state or action variable (e.g. log scale), the implementation will attempt to resolve the new variable name and prefer its values to its respective continuous definition. This greatly enhances debugging an MDP implementation by allowing for plotting the true state and action values of the MDP. An example of this custom plot is contained in Fig. 4.2

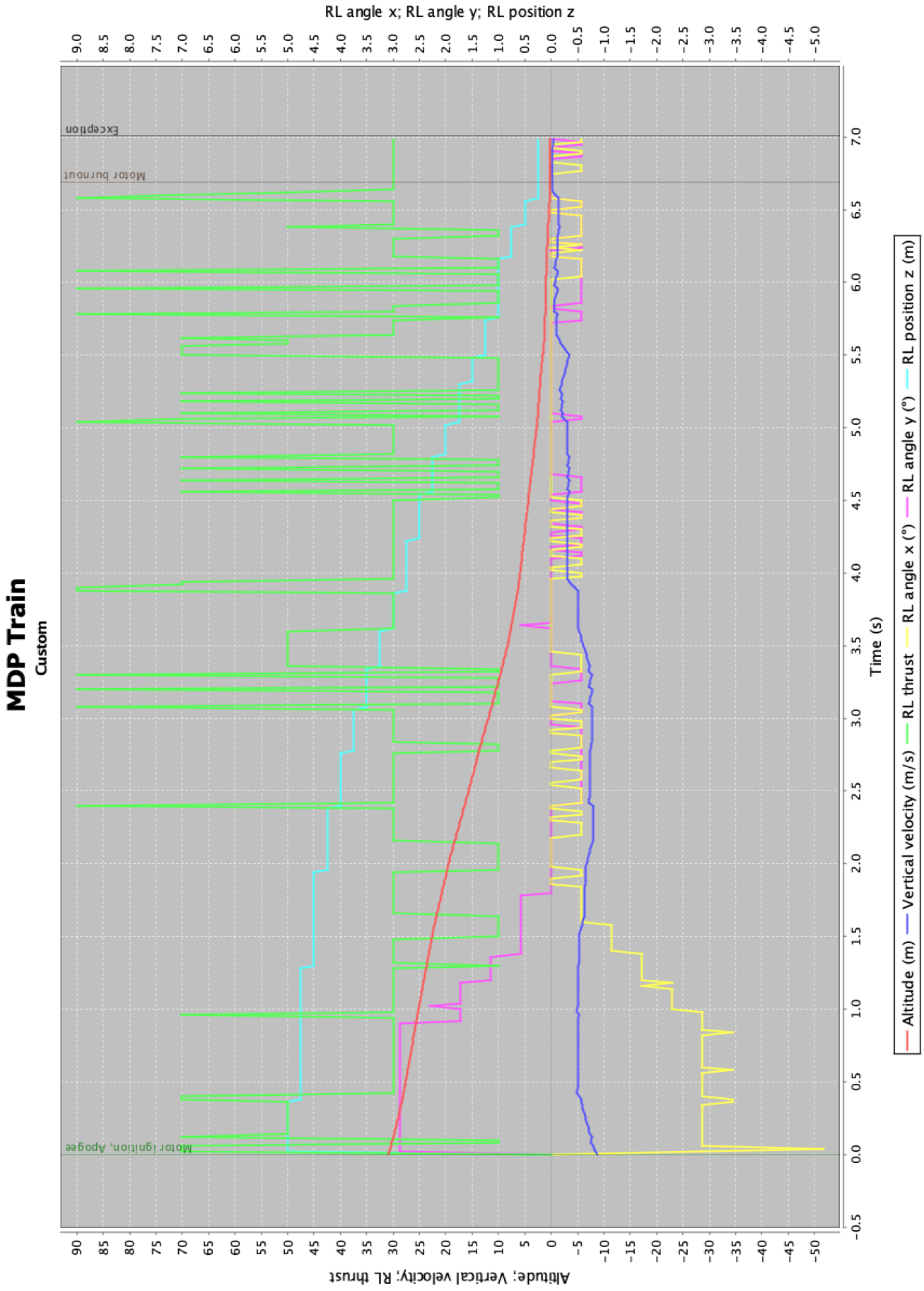



**Figure 4.2:** Example of custom plotting in OpenRocket.

# Chapter 5

# Approach: MDP Structure

This section formulates each MDP used in the final hierarchical structure and defines their state, action, and reward functions. Notably, MDPs will be discussed in reference to $x$ and $y$ axes. In the symmetry section below, I will argue that a single policy can be used for both axes. Furthermore, I use logarithmic transformations defined in equations (5.2) and (5.3) on several state variables, allowing for more fine-tuned control around critical regions.

$$sign(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ -1 & \text{if } n < 0 \end{cases} \tag{5.1}$$

$$k^{l2}(v) = sign(v) log_2(|v| + 1) \tag{5.2}$$

$$k^{l8}(v) = sign(v) log_8(|v| + 1) \tag{5.3}$$

## 5.1 Coupled Structure

Two or more MDPs are coupled when the actions of one MDP influence the state or actions of another MDP. I aim to reduce the state-space of a single monolithic MDP without reducing the number of variables or their discretization. One approach is to separate the task of controlling the thrust from controlling the gimbals into two MDPs. Then, those MDPs can be coupled on the thrust:

- **Lander**: Responsible for controlling thrust.
- **Stabilizer**: Responsible for directing the thrust.

In order to further decrease the state-space, I will assume that the task of vertical angle stabilization is independent of altitude and velocity. This assumption is reasonable only if I ignore the impact of lateral air drag.

In the following subsections, I will formulate MDPs for stabilizing the rocket's orientation, correcting the rocket's lateral velocity, and landing. Since the number of states grows exponentially with the number of dimensions, I will only include relevant states in each MDP formulation and make several assumptions to significantly reduce the size of the state space. Relevant states are those that play a key role in defining a good reward function and influence the dynamics of the MDPs' objectives.

### 5.1.1 Stabilizer

The goal of the stabilizer MDP is to correct the rocket's orientation. According to the model, this is achieved by manipulating the torque via controlling the gimbal and the thrust. According to equation (2.9), the torque is a function of the rocket's orientation and the control inputs $u$.

I want to decouple the gimbal control inputs to exploit symmetrical properties. Therefore, $u_2$ and $u_3$ are defined in the following way.

$$\sin(u_2) = -\sin(\theta_x) \tag{5.4}$$

$$\sin(u_3) = \sin(\theta_y)\cos(\theta_x) \tag{5.5}$$

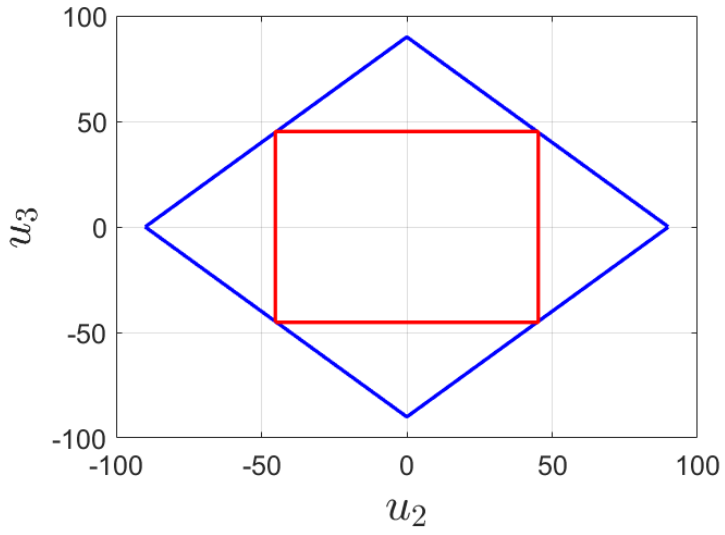



**Figure 5.1:** Bounds on feasible $u_2$ and $u_3$ (in blue) and the 45 degree limit (in red).

The gimbal angles can then be expressed in terms of the control inputs $u_2$ and $u_3$.

$$\theta_x = -u_2 \tag{5.6}$$

$$\theta_y = \sin^{-1}\left(\frac{\sin(u_3)}{\cos(-u_2)}\right) \tag{5.7}$$

Finally, the gimbal direction $d$ is redefined in the following way.

$$d = \begin{bmatrix} \sin(u_2) \\ \sin(u_3) \\ -\sqrt{1-\sin^2(u_2)-\sin^2(u_3)} \end{bmatrix} \tag{5.8}$$

As long as $u_2$ and $u_3$ are less then 45 degrees, then there exists a feasible configuration for the gimbal expressed in terms of $u$. This is illustrated in Fig. 5.1. This result is important because it allows for the MDP to chose $u_2$ and $u_3$ independently, hence, the controls are decoupled.

I want to develop a single controller that corrects the rocket's orientation by manipulating the torques $\tau_x$ and $\tau_y$ independently. These torques, however, are functions of both $u_2$ and $u_3$. With an approximation, I can treat $\tau_x$ and $\tau_y$ as functions of $u_3$ and $u_2$ only. If I assume all $sin^2$ terms are zero, which is a reasonable approximation when both the rocket angles and gimbal angles are close to zero. In the problem, small angles occur during normal operating conditions, corresponding to the rocket being close to vertical. Fig. 5.2 shows the angle difference in the true gimbal direction and the approximation over a range of rocket angles. Notably, the error introduced by the $\sin^2$ assumption is very small when the rocket is near vertical. As it flips past 45°in both rocket angles, the gimbal angle error can be more than 1°. Using results from equations (2.9), (5.11), and (5.10) and ignoring the unknown dynamics $f(x, w)$, $\tau_x$ is a function of only $u_3$ and $u_1$ and $\tau_y$ is a function of only $u_2$ and $u_1$. The approximate the gimbal forces are as follows.

$$F_x = -(\sin(u_2)\cos(\gamma_x) - \sin(\gamma_x))u_1 T(t) \tag{5.9}$$

$$F_y = -(\sin(u_3)\cos(\gamma_y) - \sin(\gamma_y)\cos(\gamma_x))u_1 T(t) \tag{5.10}$$

$$F_z = (\cos(\gamma_y)\cos(\gamma_x))u_1 T(t) \tag{5.11}$$

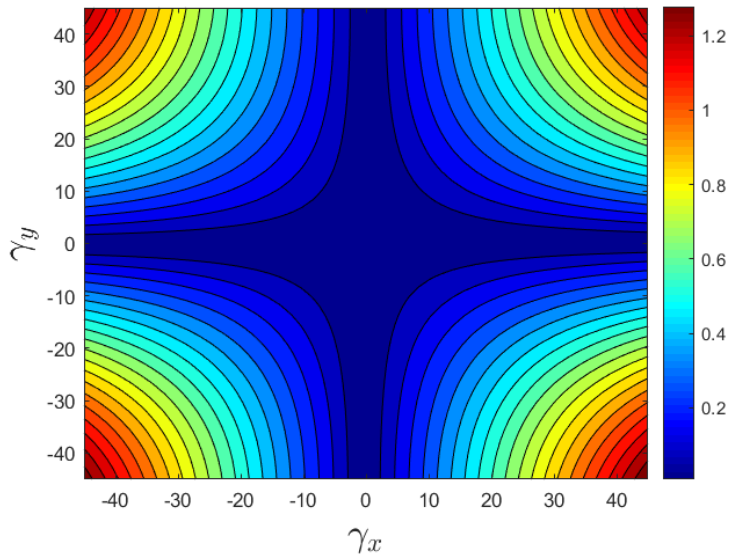



**Figure 5.2:** Gimbal angle error at different rocket angles $\gamma_x$, $\gamma_y$ due to the $\sin^2$ approximation at $\theta_x = 3°$ and $\theta_y = 3°$.

For any given thrust $u_1$, the toques $\tau_y$ and $\tau_x$ can be independently manipulated by the approximate gimbal forces $F_x$ and $F_y$.

The rocket axial direction $D$ can be expressed by rotating the $[0,0,1]^T$ vector into the global coordinate frame with $R^r$. The result is shown below.

$$D = \begin{bmatrix} \sin(\gamma_x) \\ -\sin(\gamma_y)\cos(\gamma_x) \\ \cos(\gamma_y)\cos(\gamma_x) \end{bmatrix}$$

The objective of the controller is to select torque values that minimize $D_x$ and $D_y$ and the rocket's angular velocity $\omega$.

Given these results, I define the state for the $x$ and $y$ axes of the stabilizer MDP as the following.

$$S^x = [u_1, k^{L8}(\sin^{-1}(D_x)), \omega_y]^T$$
$$S^y = [u_1, k^{L8}(\sin^{-1}(D_y)), \omega_x]^T$$

The state variables in the above definition are needed to control the dynamics of the rocket's orientation. The actions of the stabilizer MDP is defined as the following.

$$a_x = u_3$$

$$a_y = u_2$$

Finally, the reward needs to reflect the stabilizer's objective to learn a good policy. I want to minimize the rocket's non-vertical axial components, hence, I use the following reward function.

$$r(S) = 0.3 - (S_2 \frac{\pi}{180})^2$$

In summary, the stabilizer (X) controls $u_3$ with X-axis variables and the stabilizer (Y) controls $u_2$ with Y-axis variables.

### 5.1.2 Damper

In order to reduce the lateral velocity to within the goal bounds, the damper must select appropriate lateral thruster forces $u_4$ and $u_5$. Notably, the damper MDP can independently control the x

and y axis forces with the same assumption as the stabilizer: for small angles $\gamma_x$ and $\gamma_y$, the lateral thruster in the $x$ axis mainly affects $\dot{x}$ and the lateral thruster in the $y$ axis mainly affect $\dot{y}$. Therefore, the forces in global coordinates are written as follows.

$$F_x = u_4 \cos(\gamma_x) \tag{5.12}$$

$$F_y = u_5 \cos(\gamma_y) \tag{5.13}$$

Additionally, the lateral thruster forces also influence the vertical velocity depending on the rocket's orientation. The impact on vertical velocity is negligible since the lateral thrust force is an order of magnitude smaller than the main thrust force. Since $\dot{x}$ and $\dot{y}$ can be directly manipulated by the damper's action, there is no need to include states related to the rocket's dynamics in the state definition. The only states needed are the lateral velocities, which are required to craft a good reward function. The state of the damper is defined as follows.

$$S^x = [k^{L2}(\dot{x})]^T$$
$$S^y = [k^{L2}(\dot{y})]^T$$

The actions of the damper MDP are shown below.

$$a_x = u_4$$
$$a_y = u_5$$

Lastly, the reward function for the damper must penalize high lateral velocities. I define the reward function in equation (5.14).

$$r(S) = 0.2 - |S_1| \tag{5.14}$$

In summary, the damper (X) controls $u_4$ with X-axis variables and the damper (Y) controls $u_5$ with Y-axis variables.

### 5.1.3 Lander

The goal of the lander MDP is to slow the rocket's vertical decent by selecting the thrust. According to equation (2.9), the vertical acceleration is dependent on the gimbal actions $u_2$, $u_3$, and the main thrust $u_1$. To reduce the size of the state space, I will couple the lander with the stabilizer MDP. If the stabilizer selects the gimbal angles, then the lander can only influence its vertical velocity with the thrust $u_1$ according to the following equation.

$$F_z = h(S^r)T(t)u(t) + f_z(w)$$

The states that play a key role in the dynamics are time and the actions of the stabilizer and damper. To reduce the state space, I will not include their actions in the lander's state definition. The states needed to formulate a good reward function are height $z$ and vertical velocity $\dot{z}$ because a successful controller will reach the ground with velocity near zero. Therefore, I define the following state for the lander MDP.

$$S = [k^{L2}(z), k^{L2}(\dot{z}), t]$$

Since the lander only has control over the main thruster, its action is defined as the follows.

$$a = u_1$$

Finally, I need to design a good reward function for the lander. The rocket's vertical velocity on impact with the ground will be penalized according to the following terminal reward function.

$$R_t(S) = 10000(-|S_2| + 0.1) \tag{5.15}$$

Additionally, in order to have more time to recover from perturbations, the rocket should reach the ground as soon as possible. Thus, at each time step, the rocket is penalized based on time according to the following reward.

$$r(S) = -t \tag{5.16}$$

In summary, the lander controls the thrust.

## 5.2 Axial-Symmetry

The advantage of exploiting symmetry to reduce the state-space of MDPs has been presented by [17] and [18]. An explanation on how this applies to the problem is contained in the stabilizer and damper sections. The shared thrust among the x and y axes is asymmetric. However, since the stabilizer does not select the thrust, its policy is axial-independent, e.g applying a gimbal force in x manipulates orientation in x in the same way applying a gimbal force in y manipulates the orientation in y. For these same reasons, the damper also axially-symmetric. The damper's controls are subject to the same error found in the stabilizer's controls: if the rocket it tilted, the lateral thrust force in x can influence the velocity in y, and vice-versa. Given this formulation, the coupled structure can be expanded to the following MDPs:

- Lander: Controls the thrust
- Stabilizer (X): Controls $u_2$ with X-axis variables
- Stabilizer (Y): Controls $u_3$ with Y-axis variables
- Damper (X): Controls $u_4$ with X-axis variables
- Damper (Y): Controls $u_5$ with Y-axis variables

Here, stabilizer and damper are a single policy, where X and Y denote the axis on which the policy is applied on.

## 5.3 Hierarchical Structure

An MDP structure is considered hierarchical if high level MDPs are choosing among low level MDPs. I take inspiration from the options method, but with a modification. Rather than running the selected MDP until it terminates, I allow the high level MDP to choose between children MDP's at each time instant. I will formulate a high level selector MDP that chooses MDPs as its action, namely the stabilizer and damper.

### 5.3.1 Selector

I implement a selector MDP that chooses between the the stabilizer and damper in order to limit the MDP's state-space size. The selector MDP is responsible for minimizing the rocket's orientation and lateral velocity. I must stress advantage of having an additional MDP compared to adding the 'selecting' action directly to the lander, as it would require additional state variables. Notably, the lander and selector are independent, meaning they have peer relationship.

The effect of the selector's action should be influenced by the state of the stabilizer and damper. Ideally, the state definition for the selector would simply be the concatenation of the two state definitions. However, since I are trying to reduce the size of the state space, I cannot include all their combined fields. Instead, the selector's state is defined solely on the state needed to define the reward function. The problem's goal state requires that the rocket angle be vertical and the lateral velocity be minimized, thus the state is defined as follows.

$$S^x = [k^{L8}(\sin^{-1}(D_x)), k^{L2}(\dot{x})]$$
$$S^y = [k^{L8}(\sin^{-1}(D_y)), k^{L2}(\dot{y})]$$

Once again, I find myself with an MDP containing both the state variables of the x and y axes. Naturally, if the stabilizer's and damper's symmetry are both valid assumptions, then I can also defined the selector as an axially-symmetric MDP. The reward function is described in equation (5.17).

$$r(S) = ((S_1 \frac{\pi}{180})^2 + 0.3) - 10|S_2| \tag{5.17}$$

The final hierarchical structure contains the MDPs:

- Lander: Controls the thrust
- Selector (X): Selects MDP (X) with X-axis variables
- Selector (Y): Selects MDP (Y) with Y-axis variables
- Stabilizer (X): Controls $u_2$ with X-axis variables
- Stabilizer (Y): Controls $u_3$ with Y-axis variables
- Damper (X): Controls $u_4$ with X-axis variables
- Damper (Y): Controls $u_5$ with Y-axis variables

This hierarchical structure has an exponentially smaller state-space than that of a monolithic MDP, and abides by all the control constraints. The action and objective of each MDP is clearly defined in Tab. 5.1. A graphical representation of my coupled symmetrical hierarchical MDP structure is defined in Fig. 5.3.

| method | action | objective |
|---|---|---|
| lander | thrust | slow vertical landing (terminal velocity) |
| selector | MDP - select stabilizer or damper | vertical orientation and low lateral velocity |
| stabilizer | gimbal | vertical orientation |
| damper | lateral thrust | low lateral velocity |

**Table 5.1:** Actions and objectives of the different MDPs.

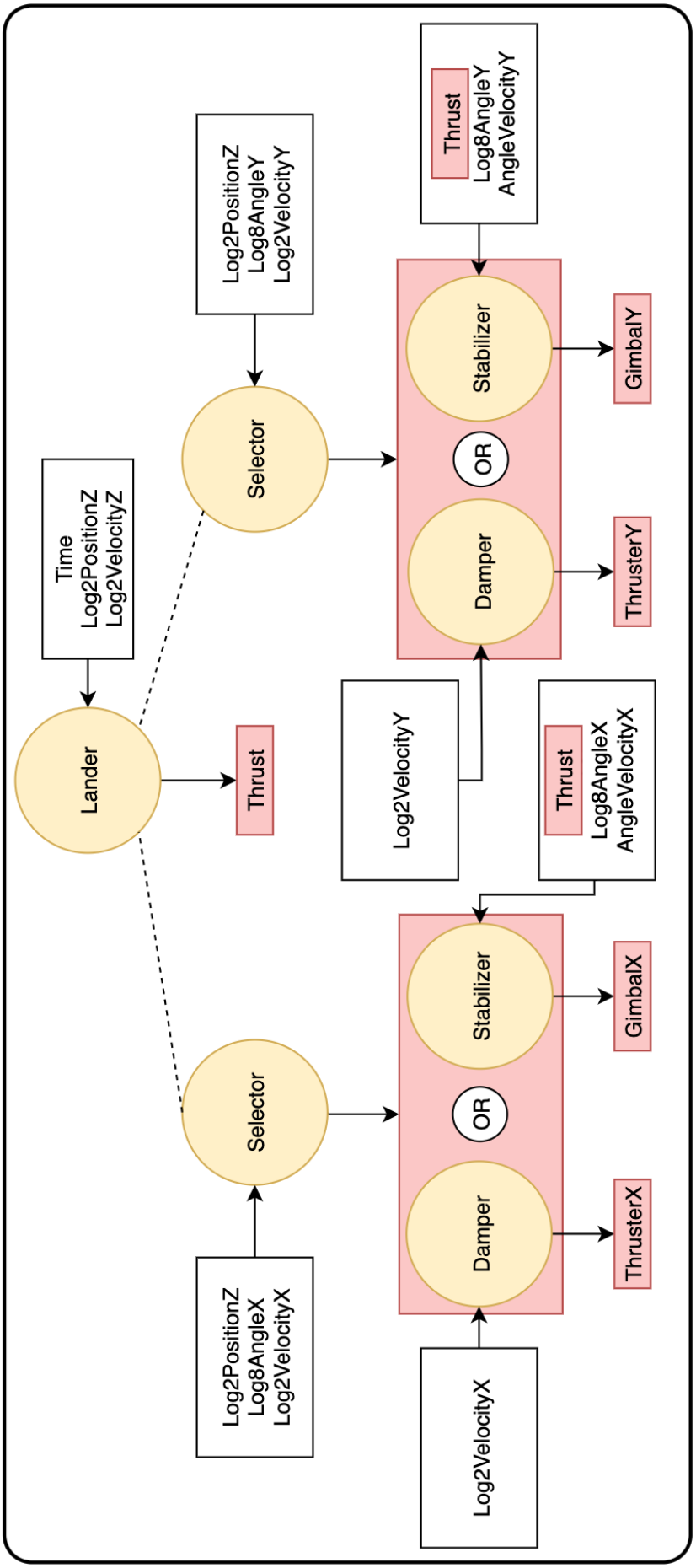



**Figure 5.3:** Hierarchical MDP structure.

# Chapter 6

# Implementation

As specified earlier, the objective involves controlling the rocket from the any initial state to any of the goal states. I present an RRT implementation as a baseline search approach and compare it to my hierarchical MDP structure.

## 6.1 RRT Implementation

As a baseline for this problem, I decided to use RRT to search for a successful state trajectory, while using the same discrete actions as the MDP formulation. This was the most similar search baseline I could build to compare my results with. The initial states of an RRT problem instance are the exact those in the problem statement section. The RRT implementation followed from the original formulation from [21], and is described in Algorithm 1. In this problem, the SELECT_INPUT

**Algorithm 1** GENERATE_RRT($x_{init}$, K, $\Delta$t)

```
Γ.init(x_init);
for k = 1 to K do
    x_rand ← RANDOM_STATE();
    x_near ← NEAREST_NEIGHBOR(x_rand, Γ);
    u ← SELECT_INPUT(x_rand, x_near);
    x_new ← NEW_STATE(x_near, u, Δt));
    Γ.add_vertex(x_new);
    Γ.add_edge(x_near, x_new, u);
return Γ;
```

method randomly samples 50 actions and forward simulates the current state $x_{near}$ for each of them, returning the action that minimizes the distance between $x_{near}$ and $x_{rand}$. The most expensive portion of RRT is usually simulating the consequences of a given action, and most importantly collision checking. Creating one node, required executing 50 simulation steps, costing approximately 1 ms CPU time. In the problem formulation, no collision checking is required, but each step of the simulator demands drastically more intensive CPU computations compared to finding the nearest neighboring node. For this reason, a KD-Tree was not implemented, which could have been used to achieve a rapid nearest-neighbor lookup.

## 6.2 MDP Implementation

After analyzing the different MDPs, I decided that two different RL methods were required, namely Monte Carlo (MC) and Temporal Difference 0 (TD0). The lander's goal is only defined at its terminal state, which is when it hits the ground at the end of the simulation, thus MC is a suitable method. The stabilizer's objective is invariant to termination time as it should always attempt to vertically stabilize the rocket and this same argument still holds for the damper's objective. The selector therefore is also suited to make decisions invariant of termination time, hence I chose TD0 as the RL method for the stabilizer, damper and selector MDPs. For my implementation, I followed the MC and TD0 pseudo-code from Sutton and Barto's Reinforcement Learning book [25] as shown in Algorithm 2 and in Algorithm 3.

**Algorithm 2** Monte_Carlo_Exploring_Starts

**Initialize:**
  $\pi(s) \in A(s)$ (arbitrarily), $\forall s \in S$
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), $\forall s \in S, a \in A(s)$
  $Returns(s,a) \leftarrow$ empty list, $\forall s \in S, a \in A(s)$
**while** another episode exists **do**
  Choose $S_0 \in S, A_0 \in A(S_0)$ randomly s.t. all pairs have probability $> 0$
  Generate an episode from $S_0$, $A_0$, following $\pi : S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  **for each** step of episode, t = $T$ - 1, $T$ - 2, …, 0 **do**
    $G \leftarrow \gamma G + R_{t+1}$
    **if** pair $S_t$, $A_t$ does not appear in $S_0$, $A_0$, $S_1$, $A_1 \ldots, S_{t-1}, A_{t-1}$ **then**
      Append $G$ to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
      $\pi(S_t) \leftarrow argmax_a\ Q(S_t, a)$

**Algorithm 3** Tabular_TD(0)

**Input:**
  the policy $\pi$ to be evaluated
**Algorithm parameter:**
  step size $\alpha \in$ (0,1]
**Initialize:**
  $V(s), \forall s \in S^+$, arbitrarily except that $V(terminal) = 0$

**while** another episode exists **do**
  Initialize $S$
  **for each** step of episode **do**
    A $\leftarrow$ action given by $\pi$ for $S$
    Take action $A$, observe $R$, $S'$
    $V(S) \leftarrow V(S) + \alpha\ [R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  **until** $S$ is terminal

The discretization of the MDPs is defined in Tab. 6.1. I used the same discretizations to create a single monolithic MDP, in which the number state-action combinations was in the order of $10^{13}$. Solving this MDP is intractable, and hence I present no results for this approach.

Fig. 6.1 contains a plot of the state fields of the hierarchical MDP structure, and allows for the analysis of the entire hierarchical system. In the first section of the graph, the selector appears to alternate the selection of the stabilizer and the damper. The second section clearly shows the selector's focus shift to stabilizing the rocket. After the angle is resolved, the selector activates the damper until the lateral velocity is near zero, and in the final section, the stabilizer is left in charge with maintaining the rocket vertical.

| | lander | selector (X/Y) | stabilizer (X/Y) | damper (X/Y) |
|---|---|---|---|---|
| **Parameters** | | | | |
| method | MC | TD0 | TD0 | TD0 |
| alpha | 0.01 | 0.01 | 0.01 | 0.01 |
| exploration | 0.01 | 0.01 | 0.03 | 0.01 |
| discount | 0.999 | 0.9 | 0.9 | 0.9 |
| **State** | | | | |
| time | [0, 9, 3] | N/A | N/A | N/A |
| $k^{L2}(z)$ | [0, 5.5, 0.25] | N/A | N/A | N/A |
| $k^{L2}(\dot{x})$ | N/A | [-2.5, 2.5, 0.25] | N/A | [-2.5, 2.5, 0.5] |
| $k^{L2}(\dot{y})$ | N/A | [-2.5, 2.5, 0.25] | N/A | [-2.5, 2.5, 0.5] |
| $k^{L2}(\dot{z})$ | [-5.0, 0.5, 0.25] | N/A | N/A | N/A |
| $k^{L8}(\sin^{-1}(D_x))$ | N/A | [-0.15, 0.15, 0.03] | [-0.05, 0.05, 0.01] | N/A |
| $k^{L8}(\sin^{-1}(D_y))$ | N/A | [-0.15, 0.15, 0.03] | [-0.05, 0.05, 0.01] | N/A |
| $\omega_x$ | N/A | N/A | [-0.21, 0.21, 0.07] | N/A |
| $\omega_y$ | N/A | N/A | [-0.21, 0.21, 0.07] | N/A |
| thrust | N/A | N/A | [0.1, 0.9, 0.2] | N/A |
| size | 2116 | 1386 | 495 | 11 |
| **Action** | | | | |
| throttle | [0.1, 0.9, 0.2] | N/A | N/A | N/A |
| MDP | N/A | stabilizer\|damper | N/A | N/A |
| gimbal (°) | NA | N/A | [-3, 3, 1] | N/A |
| lateral thrust (N) | NA | NA | NA | [-18, 18, 3] |
| size | 5 | 2 | 7 | 7 |

**Table 6.1:** Implementation details of each MDP.


**Figure 6.1:** Hierarchical policy example, where the left plot axis corresponds to $z$ (red), $\dot{z}$ (blue), zenith (green), and the right plot axis corresponds to $\dot{x}$ (pink), $\dot{y}$ (yellow).

# Chapter 7

# Results

In this section, I present the statistical results of RRT as a baseline search approach and compare it to the results of my hierarchical MDP structure. A plot showing the state-space explored by all the nodes generated by a single RRT search is shown in Fig. 7.1.

## 7.1 RRT Results

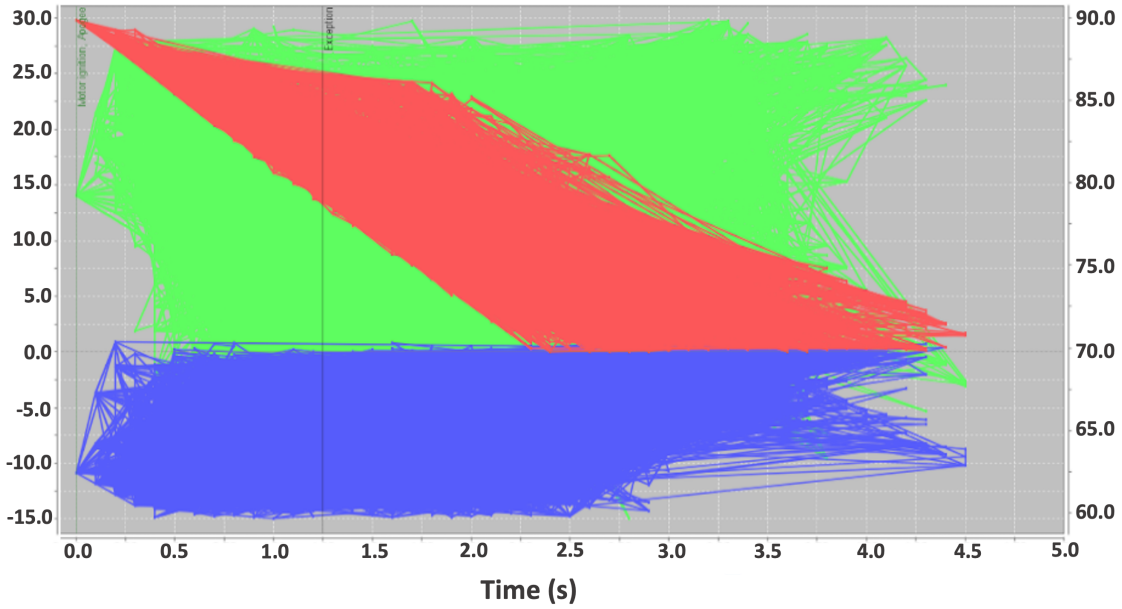



**Figure 7.1:** States explored by RRT, where the left plot axis corresponds to $z$ (red), $\dot{z}$ (blue), and the right plot axis corresponds to zenith (green).

Statistics on the number of nodes required by RRT to find a solution over 50 runs are presented in Fig. 7.2. Additionally, the minimum number of nodes expanded is 363, the maximum is 67969, and the median is 7553. Interestingly enough, the distribution appears to be geometric. A geometric distribution models the probability of an event occurring if there is a probability $p$ for each trial. For RRT, if we think of generating a node as having a constant probability $p$ of finding a solution, then the distribution of the number of nodes needed for RRT to succeed should be geometric. Furthermore, based on distribution in Fig 7.2, the probability of success for each node expansion is approximately 0.008 percent. With this insight, I triggered restarts for RRT if the number of expanded nodes exceeded 10000, since commonly a solution was found before that threshold.

## 7.2 Hierarchical MDP Results

Fig. 7.3 compares the success rate of the proposed hierarchical MDP structure to the number of training episodes. The hierarchical MDP converged on a policy with 95 percent success rate in just over 10000 training episodes. When this number is compared to the number of nodes expanded in RRT, we see that the required training is quite small. While training the hierarchical MDP,

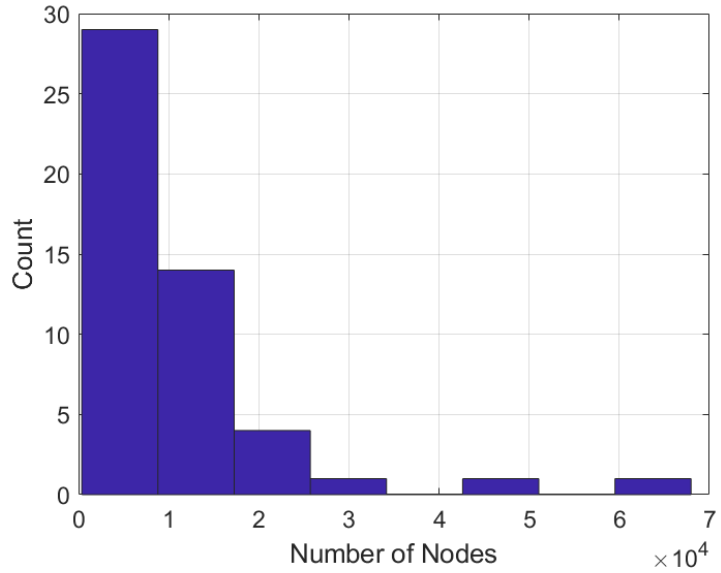



**Figure 7.2:** Number of nodes generated by RRT over 50 trials.

the average episode length is around 100 time steps. The majority of the time, RRT will find a solution after generating 2000 nodes. Generating a node involves simulating a single times step 50 times because of the SELECT_INPUT method. Hence, RRT usually requires simulating 100000 time steps before a solution is found, which is equivalent to 1000 episodes. The main benefit of my approach is that a trained policy can be executed in real time without the need for online planing. Another benefit of this approach is that the final policy is just over 50KB. Therefore, the policy can be run on a microprocessor with very limited memory.

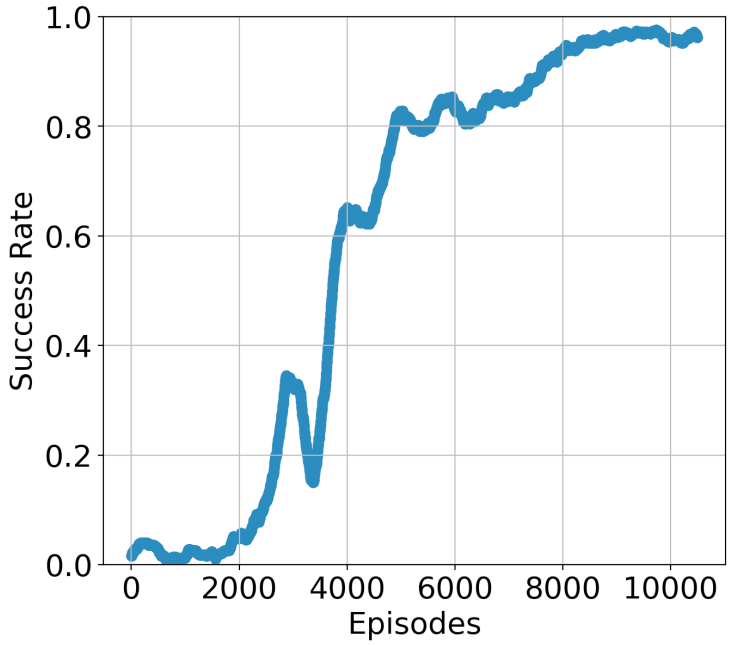



**Figure 7.3:** Success rate vs. number of episodes for the hierarchical MDP.

Videos are available on YouTube illustrating the results and showcasing the 3D visualization tool discussed earlier. The hierarchical MDP training is available here: `https://youtu.be/Zu5N9lms_EQ`. The trained hierarchical MDP is available here: `https://youtu.be/r8kyRT0Y35g`. Finally, a successful plan found with RRT is available here: `https://youtu.be/kDxorPbG7_I`.

## 7.3 Use cases of Hierarchical MDPs

When deciding between a search and an MDP approach to solve a problem, one must consider a variety of factors to make an informed selection. Several of these factors include repetitive task structure, memory limitations, and real-time constraints. Notably, the proposed hierarchical MDP structure satisfies all of these criterions. For problems such as vertical rocket-landing, where simulations are expensive, formulating a tractable hierarchical MDP by leveraging domain knowledge can be more beneficial compared to search approaches. On the other hand, search approaches are usually effective when the computation is irreducible and simulations are not computationally demanding.

# Chapter 8

# Conclusion

In this thesis, I investigated a hierarchical MDP structure for vertical rocket landing. First, I developed a framework for defining MDPs in a standardized format. With this framework, a hierarchical MDP was formulated after I developed several MDP with the sub-goals of vertical rocket landing guided by the rocket's dynamic model. The proposed structure required domain knowledge and an in-depth mathematical analysis. Additionally, I exploited coupling and symmetry to make the MDPs computationally tractable. The proposed structure was able to successfully land the rocket within the goal bounds more than 95 percent of the time. My results indicate that on-line search approaches, such as RRT may not be suitable for this application due to the high computational cost of simulating the rocket's dynamics.

In the future, I want open-source my framework and establish vertical rocket landing in a 3D environment as a standard RL benchmark. Additionally, I plan to release the OpenRocket-Blender visualization integration to the OpenRocket community, and revise the overall source code changes in order to propose the MDP definition framework as an official add-on to OpenRocket.

# Appendix A

# Appendixes

The following appendixes contain additional information relating to the development of this thesis. I start by describing the OpenRocket application in more detail and then discuss the source code implementation that was required to develop the RL algorithms present in this paper. Then I explain the UI modifications that were completed to OpenRocket, as well as the Blender visualization integration. Lastly, I included an initial formulation of the angle values that was later abandoned. Some of these appendices are in a descriptive format, while others are similar to tutorials. Notably, many appendixes contain references to Java classes, which follow the traditional naming convention in "CamelCaseClassName".

# Appendix B

# OpenRocket

OpenRocket was created for the Master's Thesis of Sampo Niskanen at Helsinki University of Technology [23], and was very impactful in the hobby rocketry community as it provided an open-source rocket simulator. Developed entirely in Java, the software package has around 47,000 lines of code. In order to implemented a self-landing rocket system, large amounts of modifications to the software were required. Throughout each stage of this project, conceded efforts were made to ensure that the impact of the changes to the core project were as limited as possible. This was achieved by implementing a OpenRocket extensions, which have the ability to obtain simulation information through a highly advanced implementation of simulation listeners. Simulation listeners have the ability to modify any parameter computation, substituting the internal logic with the modified return value. Another relevant component in OpenRocket is the SimulationStatus object. This object contains all relevant information to define the state of the rocket. Additionally, the application was implemented to be multi-threaded and multi-processed, in order to take full advantage of the hardware. Fig. B.1 contains the original OpenRocket UI.

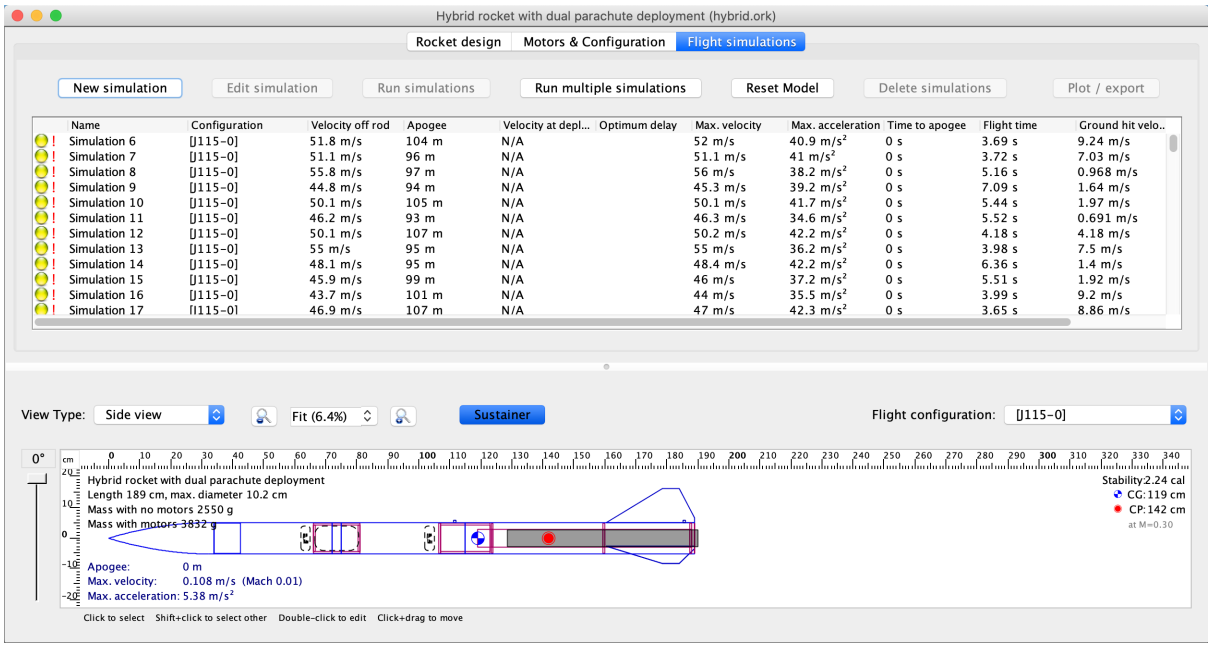



**Figure B.1:** Screenshot of the original OpenRocket UI.

# Appendix C

# OpenRocket RL Source Implementation

The following sections contains an overview of the Java implementation of the MDP framework within OpenRocket, and their titles are similar to their respective Java implementation. Notably, I only will define the classes that were most important for the objective of this thesis.

## C.1 Value Function

The standardization of the MDP definition framework relies on the MDP definitions having a value function, which is used is both the MC and TD0 RL algorithms. As defined earlier, the value function is used to develop the expected future reward of taking a specific action from a given state. Both MC and TD0 rely on a value function which is defined as an array of floats if the MDP's state-action space size is less than 1,000,000, and is otherwise defined as a HashMap of integers to floats. This implementation requires that each state be uniquely identifiable, and this is obtained by hashing the state-action tuple with an intelligent approach. Traditionally, this can be achieved by creating an $n$-dimensional matrix, where the number of dimensions $n$ is the sum of the number of state variables and action variables. In the matrix approach, each entry $1...n$ contains an array of the same size as the number of values that the $i$-th variable can assume. For my implementation, I wanted to build a single float array, and in order to achieve this I cache the number of values each variable can assume, and by ensuring their order I can construct the correct index for the 1-dimensional array.

## C.2 Custom Expressions

User-defined expressions can be inputted as strings and are converted to an Expression object which can be evaluated on a state or an action. This is achieved by accessing the state or action fields defined as variables in the expression. As said earlier, custom expressions were implemented with recursive-descent parsing by starting from a string of characters. Details on what functions can be used in expressions were defined earlier in this paper (4.3.2), and I must stress the importance of their flexibility.

## C.3 RL Algorithms

For this thesis, the RL algorithms of MC and TD0 were implemented as sub-classes of a BaseModelImplementation class. This has been completed in order to allow for more flexibility in the future. Both the MC and TD0 implementations are thread safe, and rely on the ValueFunctionManager to lock and unlock access to different indices of the value function. Notably, the selection of the RL algorithm, parameter specification and reward functions are defined in a given MDP definition.

## C.4 MDP Definition

The MDPDefinition class is the most important class for the standardized MDP definition framework. It leverages the library GSON for serialization and deserialization from and to the JSON format. This class also contains the entire logic that parses the definition object and assigns

internal values to transient fields in the instance that are used for O(1) access. Some of these "postConstructor" operations involve creating an instance of an RL algorithm and setting its parameters, as well as generating all the index bounds for each field in the "stateDefinition" and "actionDefinition" fields. Additionally, it manages equivalent state detection and the simulation termination conditions.

## C.5 Smart Plot Mapping

As discussed in the plot customization section D.5, I required an approach that was able to resolve custom field names of an MDP and tie them to the default field names. This was ensured in order to plot field names defined in an MDP definition. The DataStoreSmartPlotValues class attempts to resolve the correct field name for a given MDP structure - a set of MDPs - by starting with the lowest priority level MDP. This implementation assumes that in a hierarchical MDP the lowest level MDPs are making decisions on state or action variables that are more important than their parents' variables, and thus contain values that are more relevant for plotting. If a complete match for a default field is not defined in the stateDefinition or actionDefinition of any of the MDPs in the MDP structure, then a partial name resolution is attempted using the "String.contains" method.

## C.6 RL Model Singleton

The RLModelSingleton is a singleton class that manages all the RL related operations, from state and action creation to policy evaluation and value function updates. This class contains the implementation that recursively creates a list of states and actions for a given SimulationStatus based on the MDP definition and supports a hierarchical MDP structure. Another peculiar implementation choice is that the values for available actions are created as integer numbers, in order to avoid floating point division errors. For example, if an MDP definition's actionDefinition contains the action "thrust" defined as [0.1, 0.9, 0.2], the possible double values start from 0.1 and reach 0.9 by increments of 0.2. The double values would thus be 0.1, 0.3, 0.5, 0.7 and 0.9, and the corresponding integers for this field would be 0, 1, 2, 3 and 4. If we directly multiply the integer values by the precision, we obtain 0.0, 0.2, 0.4, 0.6 and 0.8, which is clearly incorrect. Notably, a shift is required in order to covert certain integer fields of a state or action, and this shift is pre-calculated when the MDP definition is first instantiated. In our example case, the shift would be 0.1, leading all possible action values to be 0.1 + 0.2 * 0 = 0.1, 0.1 + 0.2 * 1 = 0.3, 0.1 + 0.2 * 2 = 0.5, etc.

## C.7 State and Action Objects

Both the implementation of the State and Action classes contain a reference to their MDP definition and a HashMap mapping strings to integers. The string is the field name, whereas the integer value corresponds to the "bin number" in which the continuous value is stored with the discretization originating from the MDP definition. The bin number can be obtained from either objects by the "get(field)" method. The double value corresponding to that field is obtained by using the "getDouble(field)" method. Additionally, both classes contain a "symmetry" string field, which is used at runtime to define the symmetry of a given state or action if their definition contains the symmetryAxis parameter.

# Appendix D

# OpenRocket UI Modifications

Different OpenRocket extensions were implemented, following the pre-defined programming paradigm of extensions. Extensions are loaded into OpenRocket at runtime, requiring different files with specific class configurations to be injected into the application. The different extensions discussed below can be added to a simulation by selecting "Edit Simulation" and then entering the panel "Simulation Options". At this point, one must select "Add extension" to add the extension.

## D.1 The Abstract InitialConditions Extension

While developing the MDP definition framework, I found the need to specify the initial conditions for a problem instance in a single location that could be edited from the user interface. For this reason, I developed an abstract InitialConditions extension that contains a JSON text field defining the ranges of the initial values. The format of these entries is "field": [min, max]. Implementations of this abstract class are shown in the RocketLander and the 3DVisualize extensions. This extension is responsible for setting the initial conditions of each simulation that contain any listeners that implement the abstract InitialConditions extension.

## D.2 The RocketLander Extension

The RocketLander extension implements the InitialConditions extension and is the main class that interfaces with the RLModelSingleton class. The extension is able to observe and modify the simulation status in the pre-step and post-step phases of a single simulation step, creating state and action objects for the different MDPs when necessary. It also keeps track of the state-action tuples in its own simulation instance, and is responsible for informing the RLModelSingleton of termination of the current simulation. Additionally, the RocketLander extension contains the logic that enables thrust vectoring and applying the force of the lateral thrusters in the "preAccellerationCalculation" method. An example of the panel created when activating the RocketLander extension is present in Fig. D.1.

## D.3 Customizing Training in FlightSimulations

The FlightSimulations panel was also modified in order to run duplicate simulations in parallel, and is shown in Fig. D.2. After selecting a simulation, the button labelled "Run 500 Simulations" will duplicate that simulation 500 times and run all of them in parallel. This button is extremely useful for rapidly running many simulations with an MDP configuration. For larger training sessions, I implemented the "Extreme time training" button in the FlightSimulations panel. This was completed because I was often required to run thousands of simulations, and the most effective way was to set a temporal training threshold. An example of the options presented in the "Extreme time training" window is located in Fig. D.3.

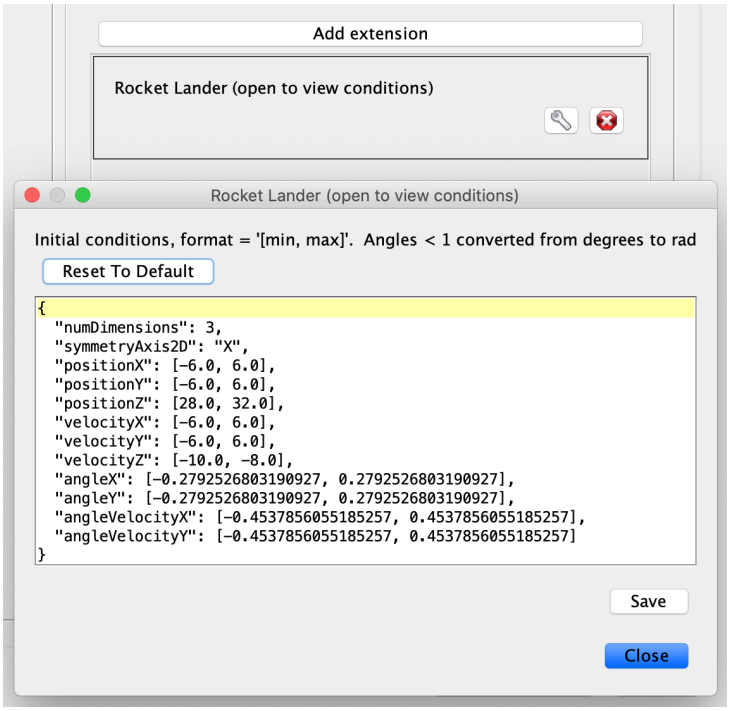



**Figure D.1:** Screenshot of RocketLander extension.

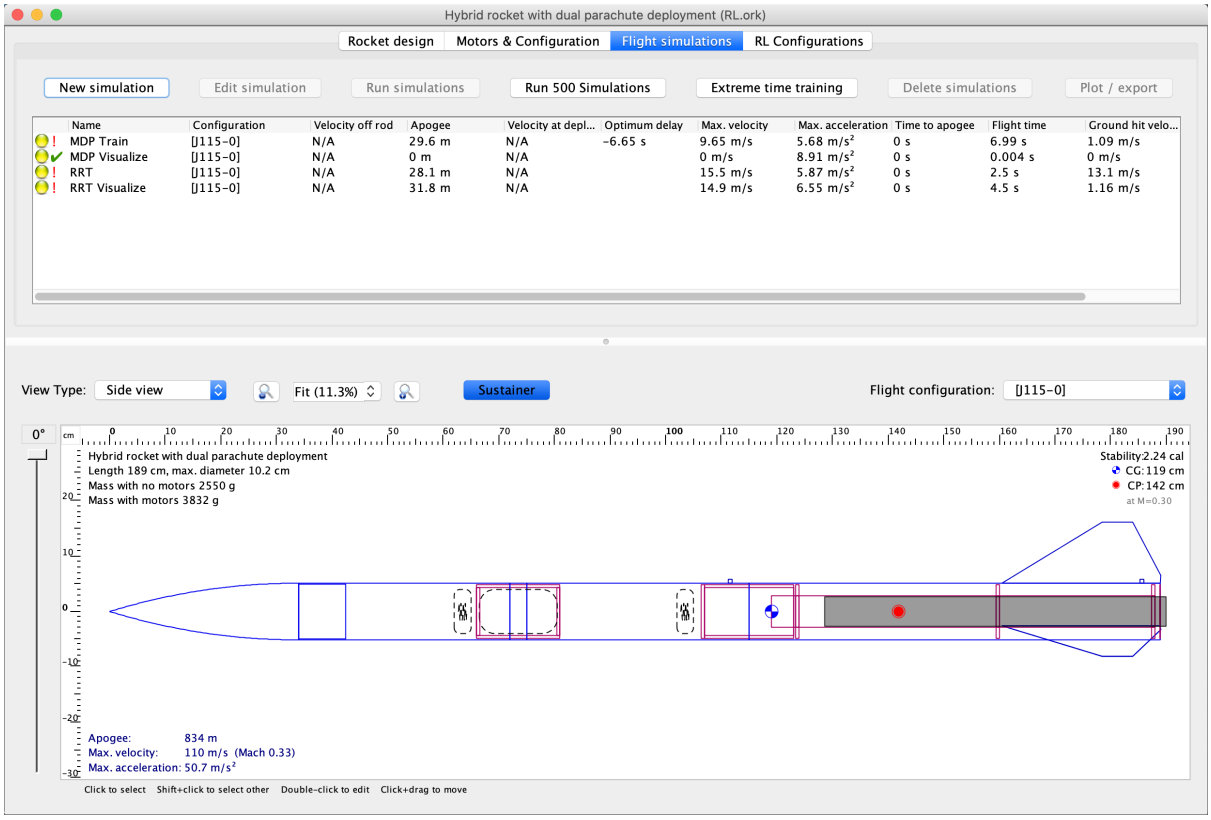
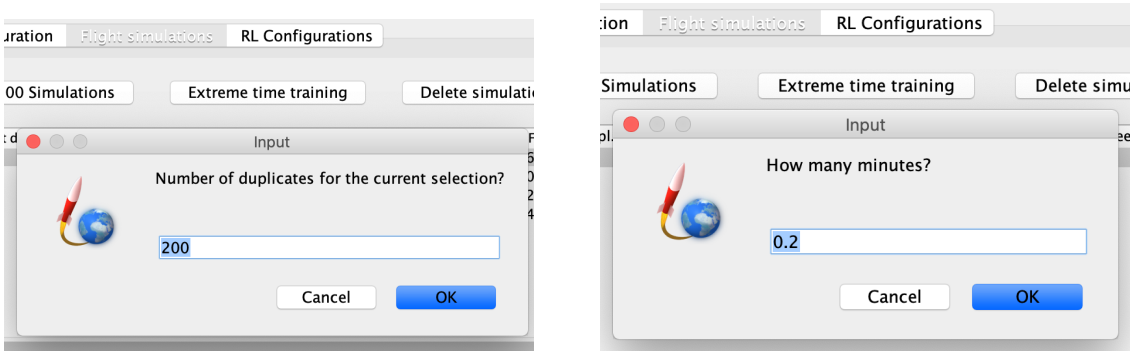



**Figure D.2:** Screenshot of the modified Flight Simulations panel.




**(a)** select number of simulations **(b)** select execution time

**Figure D.3:** Screenshots of the extreme time training option.

## D.4 The RL Configurations Panel

During the development of the standardized MDP definition framework, I needed a location in the graphical user interface (GUI) to allow for the display and modifications of the MDP definitions contained in an OpenRocket file. Thus, I created the RLConfigurations panel located in Fig. D.4. This panel allows for a standalone environment in which MDP definitions can be created, modified,

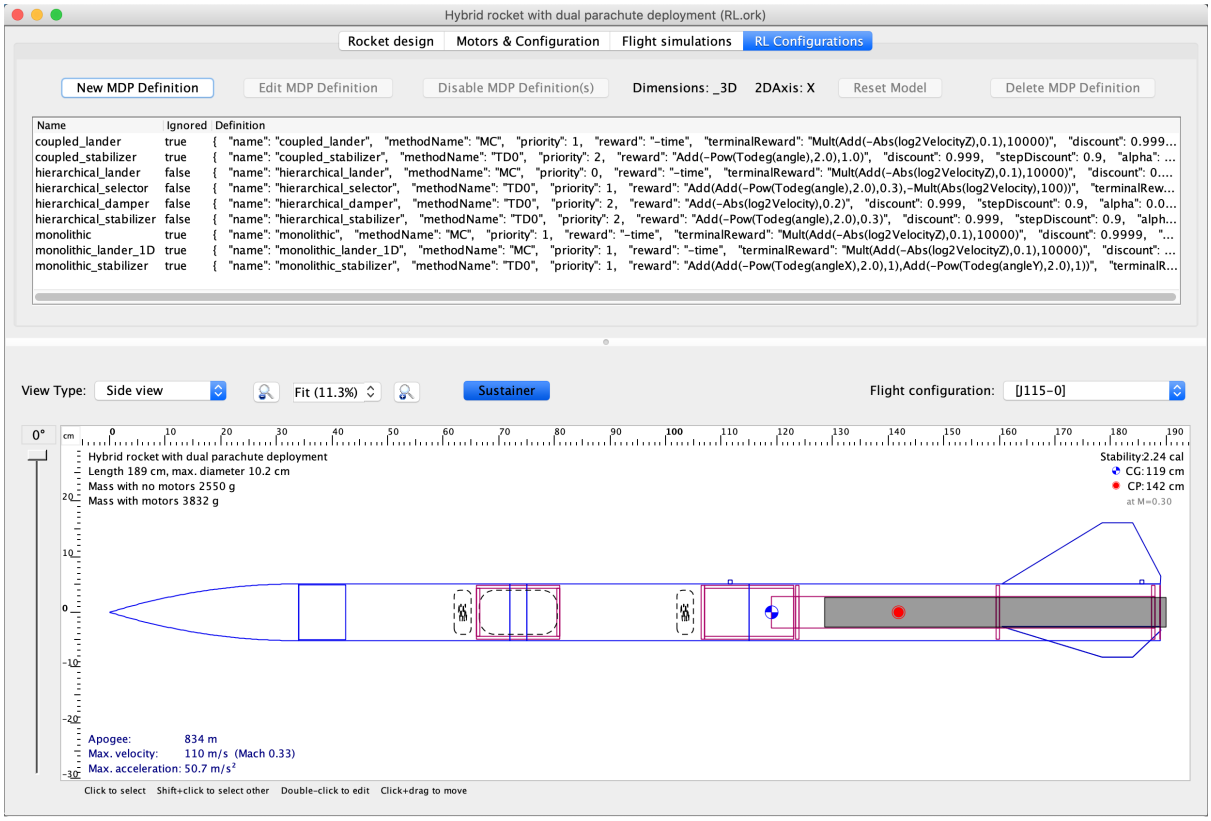



**Figure D.4:** Screenshot of the RL Configurations panel.

disabled, edited and deleted. The main component of this panel is a list of MDP definitions, which are used by the RLModelSingleton to create the required states and actions for each MDP. An MDP definition can be edited by selecting the "Edit MDP Definition" button, and one is prompted by the window shown in Fig. D.5.

While developing different MDP definitions, I also found the need to reduce the number of 'real'

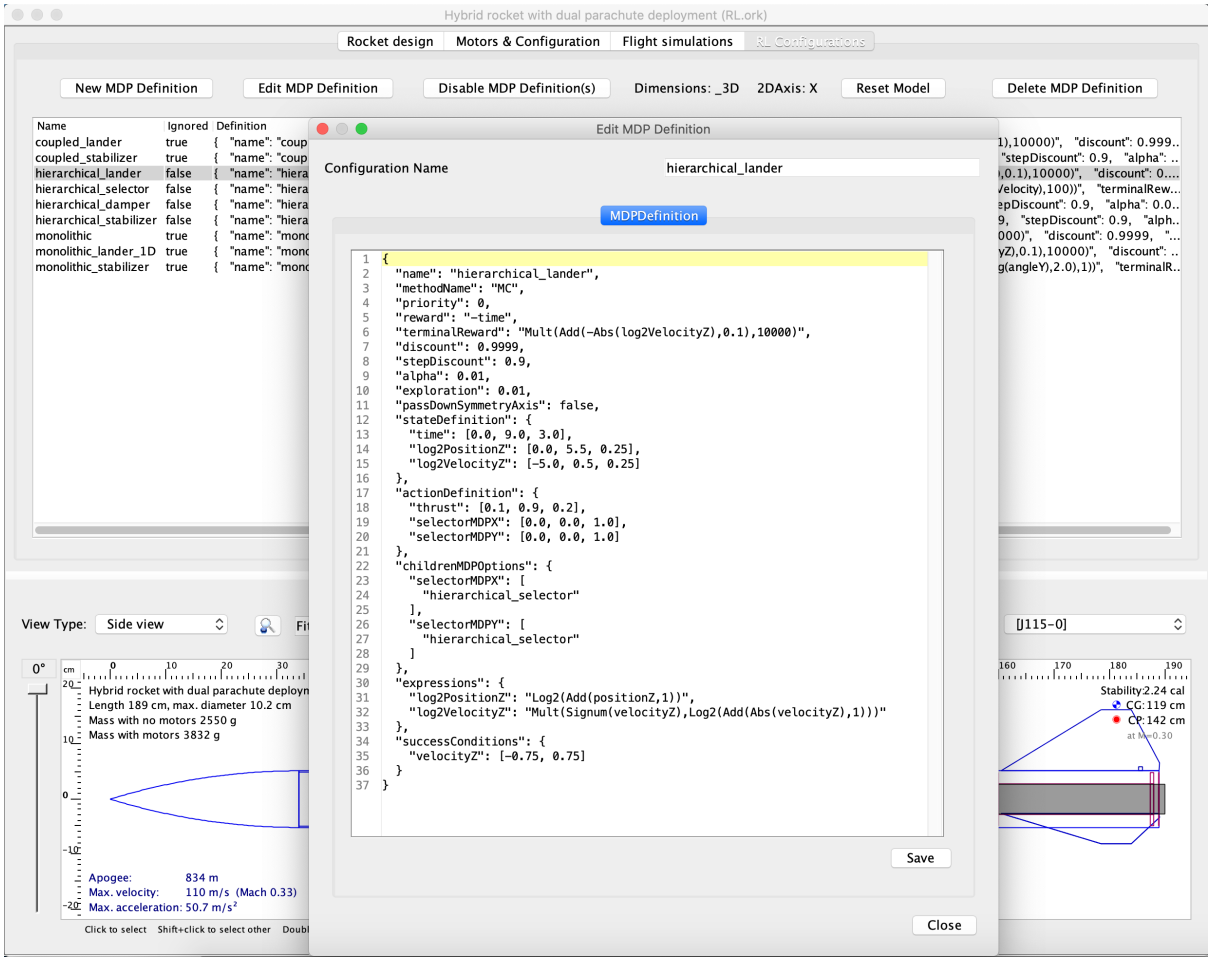



**Figure D.5:** Screenshot of editing an MDP definition in JSON format.

dimensions in which an MDP was being tested. This was extremely useful for debugging purposes, such as testing the lander in the case of a 1-dimensional landing (z axis only). Additionally, when requiring the problem to be 2-dimensional, one must decide which axis between x and y represent the secondary dimension. The number of dimensions and the 2D axis must be specified in the RocketLander extension's configurations panel (see Fig. D.1), allowing for both of these values to be displayed in the RL Configurations panel (shown in Fig. D.6).

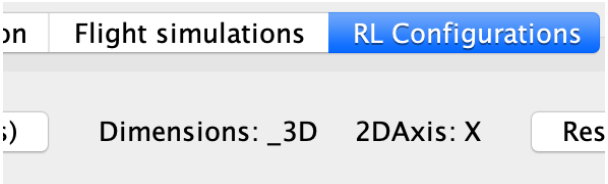



**Figure D.6:** Screenshot number of dimensions and symmetry axis for 2D.

While training RL methods, one has the necessity of resetting the model, which in our case corresponds to emptying and re-allocating the value function of a given MDP (definition). This can be achieved by selecting the "Reset Model" button in the RL Configurations panel, seen in Fig. D.7.

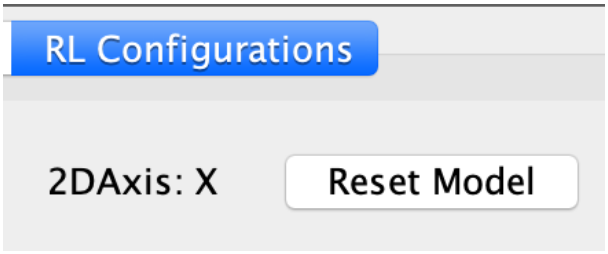



**Figure D.7:** Screenshot the Reset Model button.

## D.5 RL Plot Customization

In order to enable plotting the values contained within the state and action definitions of user-defined MDPs, I implemented a set of built-in FlightDataType objects which contain the mapped values of an MDP definition. The process of "matching" the correct state field to the correct MDP (for discretization purposes) is managed by the class RLDataStoreSmartPlotValues, which caches the user defined field names for an O(1) lookup.

### D.5.1 How to Use the Custom RL Plots

After running a simulation and selecting the built-in "Plot / export" button in OpenRocket, one is prompted with a window containing the "Preset plot configurations" drop-down as shown in Fig. D.8.

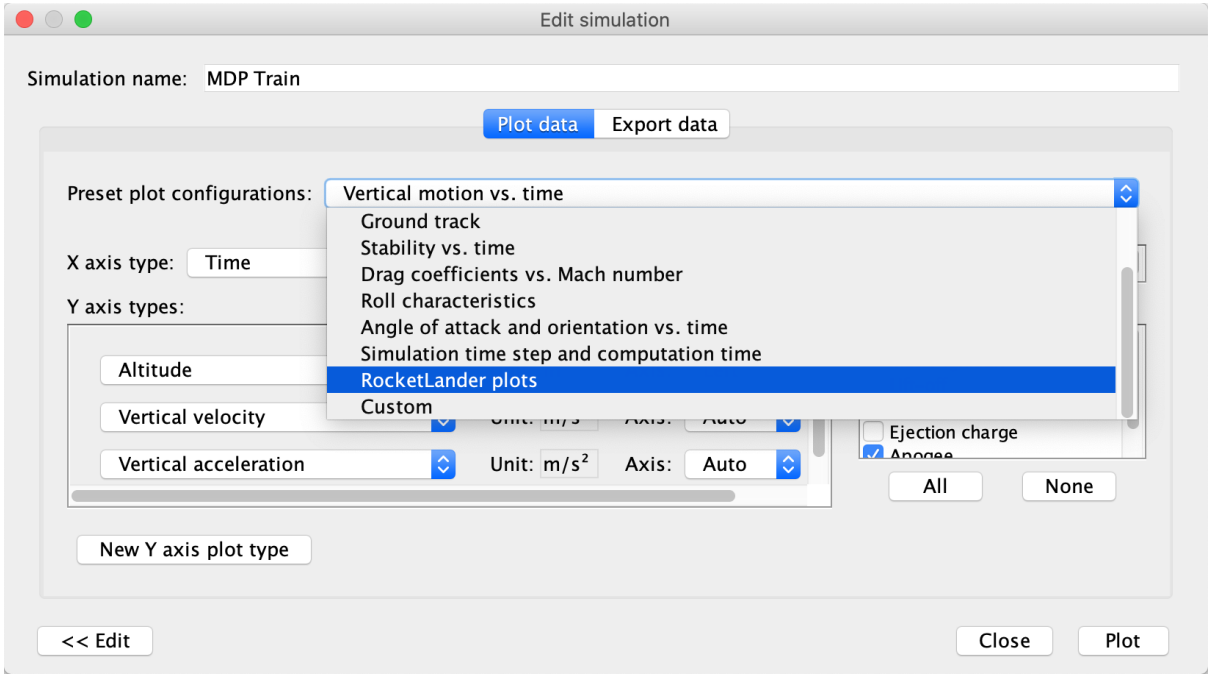



**Figure D.8:** Selecting the "RocketLander plots" configuration.

At this step, one must select the "RocketLander plots" option from the dropdown. Some state and action variables populated by the this plot configuration are shown in Fig. D.9.

An important thing to note is that the variables are forced within the MDP definition boundaries. Notably, if a custom expression is defined for a state or action variable (e.g. conversion to log scale), the RLDataStoreSmartPlotValues class will attempt to resolve the new variable and prefer its values - to their respective continuous definition - as long as the string contains one of the original field names defined in section 4.1. At this point, in order to generate a plot with these

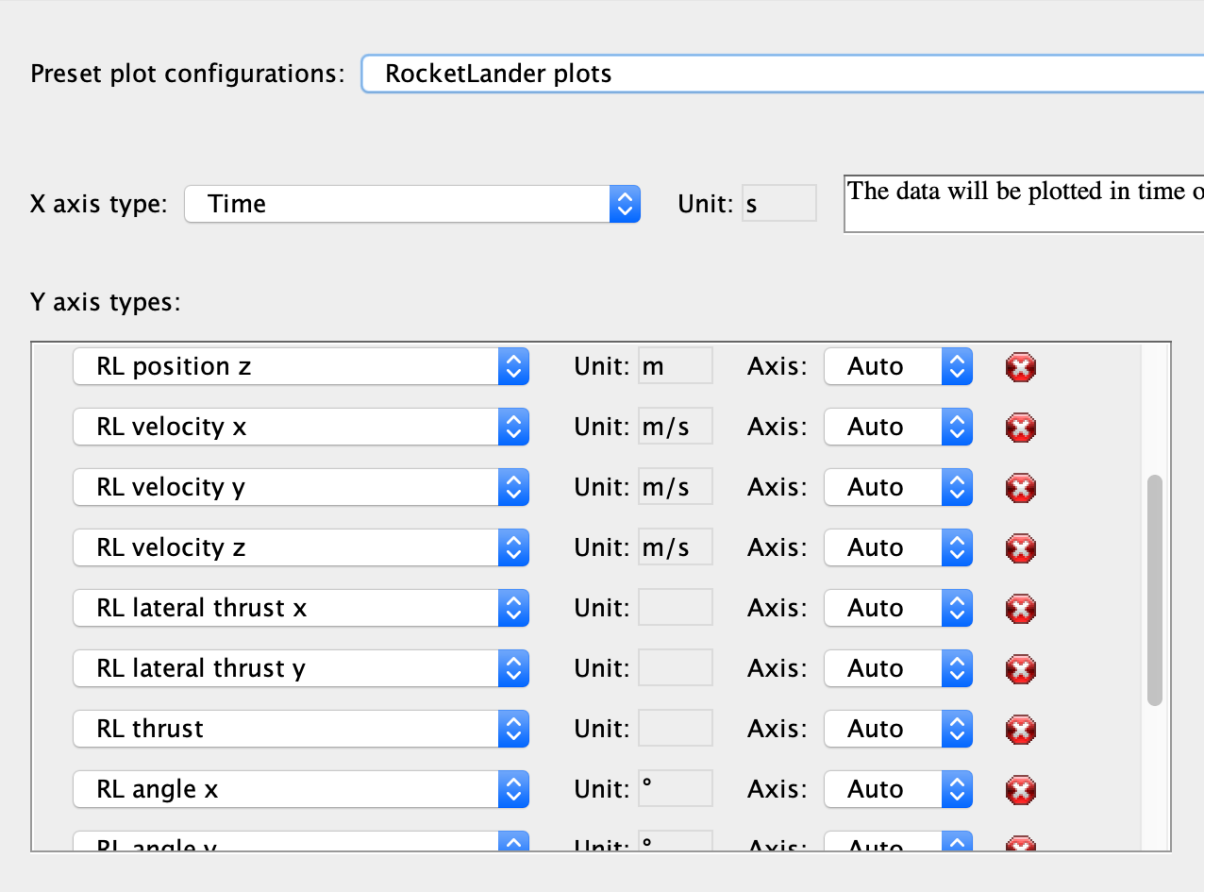



**Figure D.9:** Some variables in the RocketLander plots configuration.

fields one must select the "Plot" option. The plot will appear and contain all the available default state and action values. An example of a custom plot using the RL definitions fields is in Fig. 4.2.

# Appendix E

# OpenRocket-Blender 3D Visualization

Throughout the development of the RL methods described in this paper, I also implemented a 3D visualization extension that would empower me to understand the motion of the rocket throughout a simulation. This step was critical because it allowed to visually debug the development of the software, and ensured that the “physical” behavior of the rocket was respecting general laws of physics. I cannot stress enough the critical importance of this tool, and how greatly it impacted the development and bug-analysis. The design decision led me to select Blender as the visualization software. Blender is a free and open-source 3D computer graphics software tool used for creating animated films, visual effects, art, 3D printed models, motion graphics, interactive 3D applications and computer games. I developed a Python script that listens on a port - using a TCP server - for updates and moves the rocket throughout the 3D space, based on the information it receives on the port. In order to maximize the efficiency of the communication, I defined my own encoding format, consisting of 12 float values, representing position (x, y, z), orientation (as a quaternion), motor thrust, the gimbal angles (x, y) and the lateral thrust forces (x, y).

## E.1 Abstraction of the 3DVisualize Extension

Initially, the implementation of the 3DVisualize extension was tightly coupled with the RocketLander extension, and did not allow its use on simulations without the latter. This led to the development of the abstract "extension" class AbstractSimulationListenerSupports3DVisualizeExtension, which can be implemented by any simulation listener in order to directly interact with the 3DVisualize extension. This abstract class requires the implementation of all the action fields discussed in this paper (thrust, gimbalX, gimbalY, lateralThrustX, lateralThrustY) as well as the maximum motor thrust. By creating the abstract class, I enabled other extensions to also interact with the 3DVisualize extension, by empowering them to temporarily disable or enable the visualization process.

## E.2 How to Use the 3DVisualize Extension

I created a simulation listener for 3D visualization, which subscribes to updates of the SimulationStatus at each step of the simulation. The listener was implemented as an interface, allowing for configurable visualization preferences. The blender folder in OpenRocket contains the "linux_server.blend" file, and once this is opened directly from Blender, it loads the custom Python script. After starting the script, a TCP server is started at the local IP address with port 8080 (which is usually open on every computer and available to the local network). When running on the same machine, the address 127.0.0.1:8080 usually functions correctly (if this is not the case, it will be localhost:8080 or 0.0.0.0:8080). When running visualization on a separate machine from OpenRocket, the local IP address of that machine on the network must be identified. In unix systems, the "ifconfig" command displays the current IP address of the machine (usually under en0 or wlan0). After enabling the Visualize3D extension on a simulation in OpenRocket, and obtaining the local IP address with port 8080, the "3D View Full" tab displays the resulting rocket’s state in the simulations. Fig. E.1 contains the address and port configuration of the Visualize3D extension.

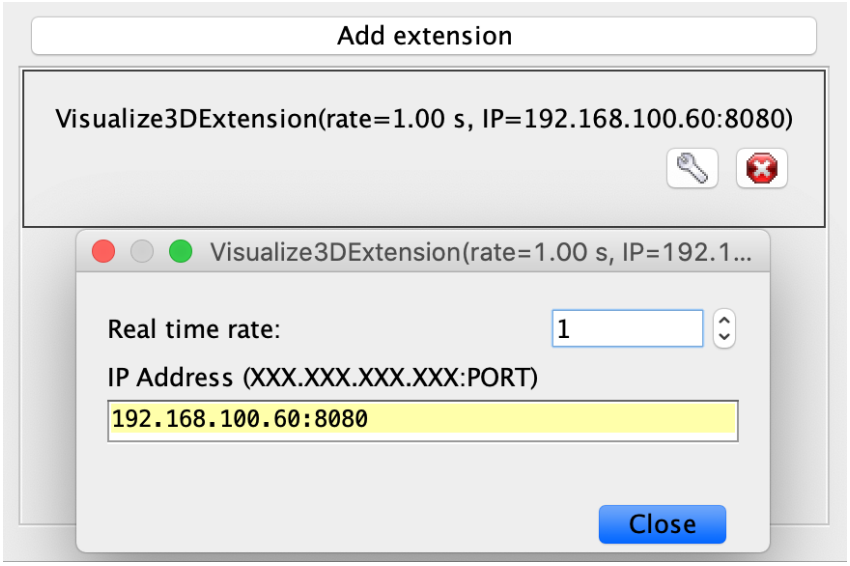



**Figure E.1:** Address and port configuration options.

# Appendix F

# Alternative Angle Definition

During an initial formulation of this problem, I attempted to reduce the variables required to define different rocket angles by mapping the 3D angle onto a 2D surface by using cylindrical coordinates. By using the angle "z" as the vertical angle from the main axis, one can define a cross section of the sphere and any angle on the cross section can be defined by an angle "y" $\in [0, 2\pi]$. This approach allows for the removal of the third angle, yet its value can be obtained by using the mathematical function atan2. This type of dimensionality reduction for the rocket's angles can be seen in Fig. F.1. This formulation was later discarded because it did not allow for a symmetrical decomposition of the angles.

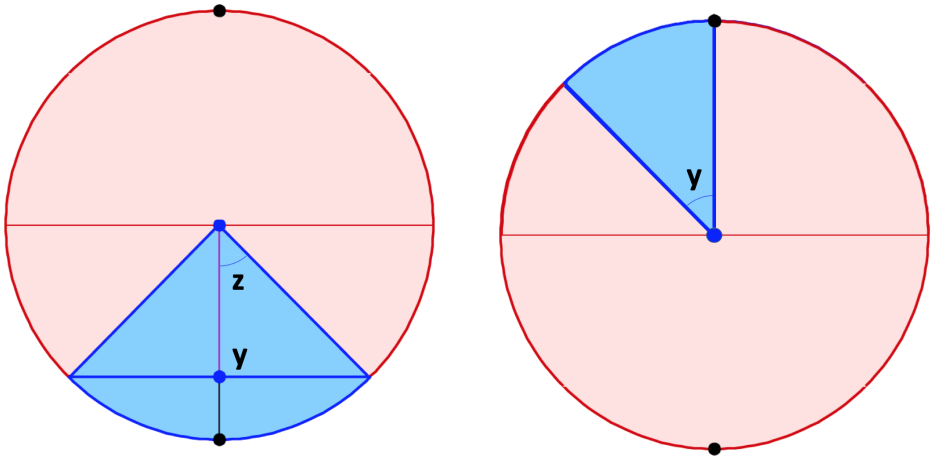



**Figure F.1:** Dimensionality reduction for the rocket state and gimbal angles.

# Bibliography


[1] R. Ferrante. «A Robust Control Approach for Rocket Landing». https://project-archive.inf.ed.ac.uk/msc/20172139/msc_proj.pdf. MA thesis. University of Edinburgh, 2017 (cit. on pp. 1, 2).

[2] T. Benson. *Brief History of Rockets*. June 2014. URL: https://www.grc.nasa.gov/WWW/K-12/TRC/Rockets/history_of_rockets.html (cit. on p. 1).

[3] National Academy of Engineering. *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2016 Symposium*. Washington, DC: The National Academies Press, 2017. ISBN: 978-0-309-45036-2. URL: https://www.nap.edu/catalog/23659/frontiers-of-engineering-reports-on-leading-edge-engineering-from-the (cit. on p. 1).

[4] G. A. Soffen and C. W. Snyder. «The First Viking Mission to Mars». In: *Science* 193.4255 (1976), pp. 759–766. ISSN: 0036-8075. URL: https://science.sciencemag.org/content/193/4255/759 (cit. on p. 1).

[5] M. G. Tomasko et al. «The Descent Imager/Spectral Radiometer (DISR) Experiment on the Huygens Entry Probe of Titan». In: *Space Science Reviews* 104.1 (July 2002), pp. 469–551. ISSN: 1572-9672. URL: https://doi.org/10.1023/A:1023632422098 (cit. on p. 1).

[6] J.-P. Bibring et al. «The Rosetta Lander ("Philae") Investigations». In: *Space Science Reviews* 128.1 (Feb. 2007), pp. 205–220. ISSN: 1572-9672. URL: https://doi.org/10.1007/s11214-006-9138-2 (cit. on p. 1).

[7] J. Taylor. *SpaceX CRS-5 Mission Press Kit*. Dec. 2014. URL: https://www.nasa.gov/sites/default/files/files/SpaceX_NASA_CRS-5_PressKit.pdf (cit. on p. 2).

[8] P. Lu. «Nonlinear predictive controllers for continuous systems». In: *Journal of Guidance, Control, and Dynamics* 17.3 (1994), pp. 553–560. URL: https://doi.org/10.2514/3.21233 (cit. on p. 2).

[9] B. Goertzel. «Cognitive synergy: A universal principle for feasible general intelligence». In: *Cognitive synergy: A universal principle for feasible general intelligence*. June 2009, pp. 464–468. DOI: 10.1109/COGINF.2009.5250694 (cit. on p. 2).

[10] J. Sutton, D. McIlwain, W. Christensen, and A. Geeves. «Applying Intelligence to the Reflexes: Embodied Skills and Habits between Dreyfus and Descartes». In: *Journal of the British Society for Phenomenology* 42.1 (2011), pp. 78–103. URL: https://doi.org/10.1080/00071773.2011.11006732 (cit. on p. 2).

[11] J. W. Krakauer. «The intelligent reflex». In: *Philosophical Psychology* 32.5 (2019), pp. 822–830. URL: https://doi.org/10.1080/09515089.2019.1607281 (cit. on p. 3).

[12] B. Cohen, M. Phillips, and M. Likhachev. «Planning Single-arm Manipulations with n-Arm Robots». In: *In Proceedings of the Robotics: Science and Systems Conference (RSS 2014)*. July 2014. DOI: 10.15607/RSS.2014.X.033 (cit. on p. 3).

[13] M. Invigorito, D. Cardillo, and G. Ranuzzi. «Application of OpenFOAM for Rocket Design». In: *OpenFOAM Workshop*. June 2014 (cit. on p. 3).

[14] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. URL: http://dx.doi.org/10.2200/S00426ED1V01Y201206AIM017 (cit. on pp. 6, 7).

[15] C. Boutilier, R. I. Brafman, and C. Geib. «Prioritized Goal Decomposition of Markov Decision Processes: Toward a synthesis of classical and decision theoretic planning.» In: *IJCAI*. 1997, pp. 1156–1163 (cit. on p. 7).

[16] S. P. Singh and D. Cohn. «How to Dynamically Merge Markov Decision Processes». In: *NIPS*. 1997 (cit. on p. 7).

[17] M. Zinkevich and T. R. Balch. «Symmetry in Markov Decision Processes and Its Implications for Single Agent and Multiagent Learning». In: *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 632. ISBN: 1558607781 (cit. on pp. 7, 20).

[18] S. Narayanamurthy and B. Ravindran. «Efficiently Exploiting Symmetries in Real Time Dynamic Programming». In: *IJCAI*. 2007, pp. 2556–2561. URL: http://ijcai.org/Proceedings/07/Papers/411.pdf (cit. on pp. 7, 20).

[19] R. S. Sutton, D. Precup, and S. Singh. «Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning». In: *Artificial Intelligence* 112.1 (1999), pp. 181–211. ISSN: 0004-3702. URL: http://www.sciencedirect.com/science/article/pii/S0004370299000521 (cit. on p. 8).

[20] T. G. Dietterich. «Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition». In: *J. Artif. Int. Res.* 13.1 (Nov. 2000), pp. 227–303. ISSN: 1076-9757 (cit. on p. 8).

[21] S. M. Lavalle. «Rapidly-Exploring Random Trees: A New Tool for Path Planning». In: 1998 (cit. on pp. 8, 23).

[22] K. Dong-Hyung, L. Sung-Jin, L. Duck-Hyun, L. Ji Yeong, and H. Chang-Soo. «A RRT-based motion planning of dual-arm robot for (Dis)assembly tasks». In: *IEEE ISR 2013*. 2013, pp. 1–6 (cit. on p. 8).

[23] S. Niskanen. «Development of an Open Source Model Rocket Simulation Software». MA thesis. 2009. URL: http://openrocket.sourceforge.net/documentation.html (cit. on pp. 10, 32).

[24] C. Runge. «Ueber die numerische Auflösung von Differentialgleichungen». In: *Mathematische Annalen* 46 (1895), pp. 167–178. URL: http://eudml.org/doc/157756 (cit. on p. 10).

[25] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018 (cit. on p. 23).