Department of Information Engineering and Computer Science

Master's Degree in
Artificial Intelligence Systems

Final Dissertation

RL-BASED OMNIDIRECTIONAL 3D JUMPING IN QUADRUPEDS

Supervisor

Prof. Michele Focchi

Dr. Giulio Turrisi

Student

Riccardo Bussola

Academic year 2023/2024

Contents

Symbols 3

Abstract 5

1 Introduction 6
- 1.1 Motivation 7
- 1.2 State of the Art 8
- 1.3 Contribution 9

2 Background 11
- 2.1 Robotic Platform 11
- 2.2 Robot Control 12
  - 2.2.1 Trajectory tracking and Gravity Compensation 14
- 2.3 Reinforcement Learning in Robotics 15
  - 2.3.1 Key Concepts of Reinforcement Learning 15
  - 2.3.2 Proximal Policy Optimization (PPO) 17
- 2.4 Simulation Solutions for Accelerating Locomotion Learning 19
  - 2.4.1 Sim-to-Real transfer 21
- 2.5 Jump Taxonomy 22
  - 2.5.1 Thrust Phase 22
  - 2.5.2 Flight Phase 23
  - 2.5.3 Landing Phase 23
- 2.6 Balistic Problem 23
- 2.7 Computing the Thrust Trajectory 26
  - 2.7.1 Bézier Curve 26
  - 2.7.2 Thrust Trajectory Parametrization 28

3 Approach 31
- 3.1 Efficient Reinforcement Learning for 3D Jumping Monopods 31
  - 3.1.1 The Action space 32
  - 3.1.2 A Physically Informative Reward Function 34
  - 3.1.3 Implementation Details 36
  - 3.1.4 Simulation Results 37
- 3.2 From Monopod to Quadruped 38
  - 3.2.1 Managing the Landing 39
  - 3.2.2 Jump Strategy for Quadrupeds 40
  - 3.2.3 Angular Extension of Thrust Trajectory Parametrization 40
- 3.3 Explosive Thrust 42
- 3.4 The new Action and State Spaces 47
  - 3.4.1 State Space 47
  - 3.4.2 Action Space 48
- 3.5 Reward Function 50

4 Results **53**
4.1 Validation of Omnidirectional Jumps 54
4.1.1 Flat Jumps and Feasible Region 55
4.1.2 Upward and Downward Jumps 56
4.1.3 Twist Jumps 58
4.1.4 Feasible Joint Trajectories 58
4.1.5 Comparation with SoA Approach 60
4.1.6 Zero-Shot on Different RoboticPlatform 62

5 Conclusions and Future Works **63**

Bibliography **64**

Nomenclature

Symbols

$\boldsymbol{\tau}$	Joint torques
$\dot{\boldsymbol{\Phi}}$	Angular velocity of the robot
$\dot{\boldsymbol{\Phi}}_{lo}$	Lift-off angular velocity of the robot
$\dot{\mathbf{c}}$	Linear velocity of the Center of Mass (COM)
$\dot{\mathbf{c}}_{lo_b}$	Bézier lift-off linear velocity of the robot's Center of Mass
$\dot{\mathbf{c}}_{lo_e}$	Explosive lift-off linear velocity of the robot's Center of Mass (UARM phase)
$\dot{\mathbf{q}}$	Joint velocities
$\boldsymbol{\Phi}$	Orientation of the robot, represented by Euler angles
$\boldsymbol{\Phi}_{lo}$	Lift-off orientation of the robot
$\mathbf{c}$	Position of the Center of Mass (COM) of the robot
$\mathbf{c}_{lo_b}$	Bézier lift-off position of the robot's Center of Mass
$\mathbf{c}_{lo_e}$	Explosive lift-off position of the robot's Center of Mass (UARM phase)
$\mathbf{F}$	Contact forces at the feet
$\mathbf{J}$	Jacobian matrix relating joint velocities to end-effector velocities
$\mathbf{K}_d$	Derivative gain in the control law
$\mathbf{K}_p$	Proportional gain in the control law
$\mathbf{P}$	Control points for the Bézier position trajectory
$\mathbf{Q}$	Control points for the Bézier orientation trajectory
$\mathbf{q}$	Joint configuration (position of all joints)
a	Action taken by the agent, corresponding to a control input
B	Base frame
d	Displacement in the UARM trajectory phase
k	Velocity multiplier in the UARM trajectory parametrization
m	Mass of the robot
r	Reward signal, quantifying how well the agent's action aligns with the task's objectives

s	State of the system, representing the environment at a specific point in time
T_{th_b}	Thrust time for the Bézier curve trajectory
T_{th_e}	Thrust time for the UARM trajectory phase
T_{th}	Total thrust time during the jump
W	World frame

Acronyms

CoM	Center of Mass.
Deep-RL	Deep Reinforcement Learning.
DoFs	Degrees of Freedom.
E2E	End-to-end Reinforcement Learning.
FK	Forward Kinematic.
GRFs	Ground Reaction Forces.
GRL	Guided Reinforcement Learning.
IK	Inverse Kinematic.
IMU	Inertial Measurement Unit.
LC	Landing Controller.
MPC	Model Predictive Control.
NLP	Nonlinear Programming.
NN	Neural Network.
PD	Proportional-Derivative.
PPO	Proximal Policy Optimization.
RL	Reinforcement Learning.
RT	Real Time.
SP	Support Polygon.
TD3	Twin Delayed DDPG.
TO	Trajectory Optimization.
UARM	Uniformly Accelerated Rectilinear Motion.
WBC	Whole Body Control.

Abstract

In this thesis, we present a novel approach for enabling real-time omnidirectional jumping in quadruped robots using guided reinforcement learning (GRL). Our work focuses on developing an efficient, safety-oriented control policy capable of handling a wide variety of jump configurations, including flat jumps, jumps with vertical displacement, and rotational jumps. The main objective is to create a control policy that maximizes jump accuracy while ensuring the robot adheres to critical system constraints, such as joint position and velocity limits, torque limits, and physical feasibility throughout the entire jump process.

A key contribution of this thesis is the introduction of a two-stage thrust trajectory parametrization in task-space, combining the flexibility of Bézier curves for the initial jump phase with a Uniformly Accelerated Rectilinear Motion (UARM) model for the final more explosive phase. This hybrid approach enables precise and dynamic control of both linear and angular components of the jump, allowing for complex aerial maneuvers such as mid-air orientation adjustment and twist jumps, while maintaining low action space complexity.
To ensure efficient and scalable training, we leveraged state-of-the-art parallelization techniques through the Isaac Sim and Orbit frameworks, simulating thousands of robots in parallel.

We evaluated the proposed framework through extensive simulation tests, demonstrating that the policy can perform highly accurate jumps up to 0.6 meters in distance, with vertical displacements reaching 0.26 meters upward and 0.4 meters downward. The policy also successfully handled jumps requiring orientation adjustments, maintaining low angular error and demonstrating effective control during dynamic movements. The learned policy proved to be highly sample-efficient, achieving comparable or superior performance to state-of-the-art end-to-end (E2E) approaches with significantly fewer training samples.
When compared with the multi-stage E2E method by Atanassov et al., our approach demonstrated a key advantage: while their method enables longer jumps, it often compromises system constraints. In contrast, our approach prioritizes safety and physical feasibility, ensuring that all jumps respect the robot's physical limits.

Additionally, we validated the generalization capability of the policy by performing a zero-shot transfer to a different quadruped platform, from the Unitree Go1 to the Unitree AlienGo. With minimal adjustments, the policy successfully executed jumps on the AlienGo platform, showcasing the adaptability of the proposed approach across platforms with similar characteristics.

This work establishes a robust foundation for advancing quadruped locomotion in challenging environments, with promising applications in exploration, search-and-rescue, and industrial inspection automation.

1 Introduction

From science fiction narratives to contemporary technological advancements, robots represent a fascinating concept and a challenging engineering problem. Although a precise definition is lacking, robots are most accurately described as systems capable of perceiving their environment and autonomously interacting with it accordingly. Recent advancements in artificial intelligence have made the applications of robotics increasingly intriguing and closer to the visions depicted in science fiction.

Robots can be categorized into two classes: fixed-based and mobile robots. While fixed-based robots are constrained to a stationary base, mobile robots are free to move, offering a broader range of applications. Mobile robots can be deployed to assist or substitute humans in environments where movement is required, when there are payloads to transport, and especially in situations where human presence might be at risk. Tasks for mobile robots include exploration, search and rescue operations, and autonomous inspection. In exploration missions, they can navigate and map unknown or hazardous environments, gathering valuable data for scientific research or resource management. In search and rescue operations, mobile robots can access disaster areas that are unsafe for human rescuers, locating survivors and providing critical assistance. Autonomous inspection involves monitoring infrastructure such as bridges, pipelines, and industrial facilities, where robots can detect anomalies or defects, thereby preventing accidents and enhancing safety protocols.

Mobile robots can be categorized furthermore into two types: wheeled robots and legged robots. While wheeled robots may offer higher speeds due to their wheel-based locomotion, they are generally unsuitable for operating in unstructured terrains that present various impediments, such as stairs or rubble. Legged robots, on the other hand, are capable of stepping over obstacles, climbing, and even performing aerial maneuvers (e.g. jumping).
Although bipedal robots (humanoids) provide platforms that can easily interact with environments designed for humans, they are extremely difficult to balance and, consequently, to control. Quadruped robots, designed to mimic quadrupedal animals, offer greater stability due to a larger Support Polygon (SP) (the convex hull formed by the robot's contact points with the ground). This increased stability makes quadrupeds particularly well-suited for dynamic environments where conditions can change rapidly and unpredictably. They can adapt to uneven terrains, maintain balance during sudden movements, and recover from disturbances more effectively than bipeds. The distributed support of four legs allows for more complex gait patterns, enabling them to navigate challenging terrains and respond quickly to environmental changes.

Reinforcement Learning (RL) policies are being trained to extend the locomotion and manipulation capabilities of legged robots[1]. For years, quadruped platforms have been the subject of extensive research [2], and recently, an increasing number of such robots have been introduced to the market for both research institutions and private industry (see Fig. 1.1)[1]. This trend demonstrates a growing demand for solutions capable of operating effectively in dynamic and unstructured environments.

[1]Images sources: ANYbotics ANYmal `https://www.anybotics.com/robotics/anymal/`, Boston Dynamics Spot `https://bostondynamics.com/solutions/inspection/`

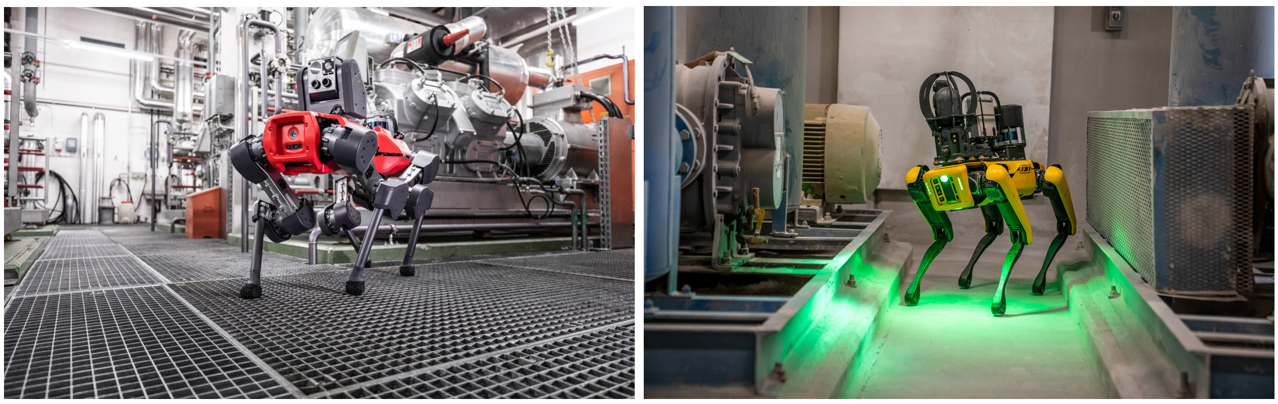

(a) ANYmal from ANYbotics (b) Spot from Boston Dynamics

Figure 1.1: Two of the most popular commercial quadruped robots equipped with sensors for autonomous inspection in industrial environments

1.1 Motivation

While considerable advancements have been made in locomotion strategies for walking and running [3], the more challenging task of jumping remains a relatively unsolved problem in quadruped robotics. Innovative parkour strategies, such as those proposed in [4], aim to fill this gap but rely heavily on intensive RL training in simulation and are purely End-to-end Reinforcement Learning (E2E) approaches. The ability to jump is essential for overcoming obstacles that exceed the reach of conventional gait patterns and surpass the maximum leg extension length.
Tasks that require in-place or omnidirectional jumping with changes in orientation include navigating complex urban environments with debris or gaps and maneuvering in disaster zones where the terrain is unpredictable during search and rescue missions. For example, a quadruped robot may need to jump onto elevated platforms, change orientation mid-air to fit through narrow openings, or rapidly reorient to avoid falling debris.
Although jumping is intrinsic in the animal kingdom, it is a dynamic action that requires the system to operate at the limits of its performance capabilities to achieve the desired goal. One of the primary challenges in quadruped jumping lies in developing a control strategy that pushes the robot to its performance limits without causing damage.
The complex interaction between the robot's dynamics and the unstructured nature of real-world scenarios can result in small environmental variations leading to drastic changes in the robot's performance during a jump [5]. These variations impose significant limitations on the robot's ability to predict and control its movement, making the development of robust jumping strategies even more challenging.
Jumping involves a brief initial contact with the ground, limited to the takeoff phase. During this very short period, the robot cannot adequately cope with disturbances, and the success of the jump depends directly on the configuration at lif toff. In its airborne phase, the robot has limited ability to adjust its orientation and it cannot provide additional thrust, as the outcome of the landing is governed purely by the conservation of momentum. As a result, even minor deviations in takeoff conditions or environmental factors can lead to significant discrepancies in landing position and orientation.
Exploring the limits of quadruped jumping capabilities is essential for enhancing the versatility and adaptability of these robots in dynamic environments. By studying and addressing this challenge, we can develop more sophisticated control algorithms involving RL strategies and discover new ways of fusing model-driven approaches with purely data-driven ones. RL serves as a crucial tool for addressing modeling inaccuracies and complexities, reducing the computational time and power required to handle dynamic strategies that traditional methods cannot solve efficiently.

These advancements will enable quadruped robots to perform complex maneuvers with greater reliability, thereby expanding their utility in applications that demand high levels of mobility and resilience (i.e. RL for jump can be introduced in system like the climbing robot proposed by Focchi et al. [6]). This work aims to provide an innovative solution to fill the existing gap in the field of quadruped jumping. While traditional legged locomotion enables quadruped robots to traverse a wide range of terrains, it becomes inadequate when facing obstacles or gaps larger than the robot's leg span. In such scenarios, jumping offers a critical advantage, allowing the robot to overcome obstacles and gaps that cannot be negotiated through standard walking or running gaits. This capability enhances the robot's versatility and operational range, particularly in complex environments such as disaster areas or rugged terrains, where mobility is crucial and human intervention is either too risky or impractical.

By addressing the limitations of quadruped jumping, this work not only advances the state of the art in robotics but also provides practical solutions for real-world applications, improving the ability of robots to navigate challenging environments with greater autonomy and efficiency.

1.2 State of the Art

The challenging problem of quadruped jumping has garnered increasing attention in recent years, with researchers proposing various approaches to address it. When employing a numerical optimization approach, the primary challenges include the time-intensive computations needed to generate feasible jump trajectories, the necessity of comprehensive full-system modeling, accurately capturing dynamic contact timing, and the inherent sparsity of the task, given that control is primarily concentrated in the thrust phase. These complexities make it difficult to efficiently compute solutions, especially when the system's dynamics and contact interactions must be precisely accounted for.

Optimization-based methods have traditionally employed reduced models, such as a 5 Degrees of Freedom (DoFs) representation of the quadruped's side view within the sagittal plane, along with heuristic approaches that utilize physical intuition for designing controllers or planners [7, 8]. While these hand-crafted motion actions are grounded in physical reasoning, they often lack guarantees of physical feasibility. Another common approach involves full-body numerical optimization [9, 10], which has demonstrated impressive results, such as the MIT Mini Cheetah's [11] ability to perform complex aerial maneuvers like jumps, spins, flips, and barrel rolls using a centroidal momentum based nonlinear optimization [12, 13, 14]. However, these optimization-based methods for high-dimensional nonlinear problems are computationally expensive, making them impractical for Real Time (RT) applications, particularly for online trajectory re-planning.
While Model Predictive Control (MPC) has seen significant improvements for jumping tasks [15, 9], its usage often comes with trade-offs, such as the need to impose artificial constraints, like pre-fixed contact sequences, time-of-flight, or offline-optimized timing parameters. Pre-specified foot contact can lead to instability issues in the presence of large mismatches between the expected and actual contact points [16]. A novel approach by Bellegarda et al. [17] addresses these limitations by rapidly optimizing parameterized foot force profiles online, reducing reliance on pre-specified constraints.

Recently, RL has emerged as a promising approach to address the limitations of traditional optimization methods. Advances in computational power, time-continuous action algorithms, and highly parallelized simulators have spurred interest in applying RL to robot locomotion tasks. The pioneering work by Lillicrap et al. [18] demonstrated that actor-critic mechanisms combined with deep-Q networks can successfully learn policies in continuous action domains. Building on this foundation, RL has been applied to various quadruped locomotion tasks [19, 20, 21, 22, 23] and even early attempts at in-place hopping with legs [24].
However, one major drawback of RL is the requirement for an enormous number of training steps, often in the millions, to converge on effective solutions, as highlighted by OpenAI's benchmarks [25]. To improve the efficiency and robustness of glsrl training, several approaches have proposed combining

Trajectory Optimization (TO) with RL [26, 27, 28]. In these methods, RL is used to generate initial trajectories that bootstrap TO's exploration process, addressing issues of local optimality and improving convergence. The action space design is another crucial factor in RL efficiency and robustness [29, 30]. Depending on the approach, controllers may generate position or torque references [31] or operate in Cartesian or joint space [32]. A notable generalization of RL across different quadruped morphologies has been proposed by Shaffie et al. [33], where both the action and observation space are formulated in task space, making it independent of the robot's intrinsic characteristics and morphology.

Despite the growing interest in RL for quadruped locomotion, its application to the jumping task has received relatively little attention. Unlike walking or running, jumping involves a prolonged airborne phase, making it a sparse reward problem (rewards are only accrued at the conclusion of the action upon touchdown). This sparse reward landscape complicates learning through conventional RL approaches [34], as the abrupt changes in contact during jumps create non-smooth reward functions, making convergence to optimal policies difficult. Among the few successful RL applications to the jumping task, Yang et al. [35] use a warm-start strategy by learning residual actions on top of a controller, which reduces reward landscape noise and improves training convergence. However, their work focuses solely on fixed-duration jumps. Vezzi et al. [36] proposed a multi-stage learning approach aimed at maximizing jump height and distance, but it gives little attention to jump accuracy.
Atanassov et al. [37] achieved remarkable results with a complex multi-stage training process and heavily engineered reward shaping. Although their work represents the state-of-the-art in RL-based jumping, the complexity of the approach may limit its general applicability. For this reason, we will use their work as a baseline to compare with the methods proposed in this thesis.

A comprehensive overview by Esser et al. [38] suggests that incorporating task-specific knowledge (known as Guided Reinforcement Learning (GRL)) is the most effective way to address the challenges posed by E2E in complex tasks. Our previous work on monopod jumping [39] provides valuable insights into how GRL can be used to address the sparsity and complexity of the linear jumping task, forming the foundation for our approach to quadruped jumping in this thesis.
In conclusion, while optimization-based and RL methods have made significant strides in quadruped locomotion, there is still a clear gap in effective solutions for the quadruped jumping task, particularly in non flat environments.

1.3 Contribution

This work proposes the GRL-based strategy for precise 3D omnidirectional quadruped jumping, designed to plan the entire jumping action in just a few milliseconds, thereby enabling RT applications. This is achieved by parameterizing the jump trajectory in Cartesian space, which allows us to model the jumping task as a single action, thus addressing the sparsity issue inherent in conventional RL approaches. Instead of relying on a sequence of actions, the entire jump is now encapsulated in one action, significantly simplifying the problem.
By incorporating physical knowledge to model the jump trajectory, our approach not only enhances accuracy but also ensures safe predictions of the jump outcome. Given the robot's lift-off configuration, we can use ballistic motion equations to accurately predict the landing position. This predictive capability adds a high degree of reliability to the glsrl policy, ensuring safer and more dependable actions. This research extends our previous foundational work on monopod jumping [39] to the more complex quadruped scenario, with the main advancements involving the inclusion of angular dynamics, which were previously neglected.

By integrating both orientation and angular velocity into the model, the robot can now fully exploit its dynamic capabilities. This enables the robot to optimize its orientation at takeoff, shaping the forces distribution generated during the thrust phase. Additionally, controlling angular velocity allows the robot to adjust its pitch mid-air, reducing impact forces upon landing. Furthermore, this approach enables novel jump maneuvers, such as in-place twist jumps, where the robot can rotate its body in mid-air without linear displacement, expanding its repertoire of dynamic actions. Planning the action in Cartesian space also makes the system adaptable to different quadruped morphologies (or even the robot morphology, e.g. humanoids).

The method computes Center of Mass (CoM) references, which are then mapped to joint positions using Inverse Kinematic (IK). As long as you can provide the necessary IK, this system can be applied to different robot designs. The joint references are tracked by a low-level controller, leveraging the robustness of classical control methods to handle uncertainties such as mass variation or other imperfections that arise during the sim-to-real transition. This removes the need for extensive domain randomization during the training process, making our method more robust.

Unlike other approaches that require multi-stage learning or preliminary goal-specific tasks, our method is a single-stage learning process that directly achieves the final goal of jumping without the need for intermediate training steps. This simplification reduces complexity, improving the sample efficiency and training time while enhancing performance.

2 Background

2.1 Robotic Platform

The target robotic platform used to develop and demonstrate the effectiveness of our innovative jumping method is the Go1 quadruped from the Unitree company [40]. This model is a widely recognized consumer-level platform, especially popular for research and development in legged locomotion. Its lightweight yet robust design allows for enhanced agility and adaptability, making it an ideal solution for applications in robotic research. The SDK, robot description, and various software packages for operating the Go1, as well as other Unitree robotic platforms, are available through the company's GitHub profile [41].

The Go1 is a quadruped robot featuring four identical legs symmetrically distributed around a central body link known as the trunk, to which all legs are connected. Each leg consists of three link (hip, thigh, and calf) connected by the following joints:

- **Hip Abduction-Adduction** (HAA) with a range of $[-49°, 49°]$
- **Hip Flexion-Extension** (HFE) with a range of $[-39°, 257°]$
- **Knee Flexion-Extension** (KFE) with a range of $[-161°, -51°]$

Therfore, each leg has 3 DoFs, with each joint powered by a high-torque GO-M8010-6 permanent magnet synchronous motor. The knee joint motors, which commonly bear the greatest load, are equipped with a 1.5:1 gear ratio and can generate up to 35.55 Nm of torque. In contrast, the hip and thigh joints operate without a gear ratio and can deliver up to 23.7 Nm of torque. In total, the robot has 12 actuated DoFs. As we will see in the control section, legged robots, including this platform, are considered underactuated systems due to their floating base nature.

Each leg of is labeled according to its position relative to the robot's body:

- **FL** : Front Left leg
- **FR** : Front Right leg
- **RL** : Rear Left leg
- **RR** : Rear Right leg

The robot is equipped with a 6-DoFs Inertial Inertial Measurement Unit (IMU) sensor, and each foot is embedded with a force sensor.
The Go1's dimensions in its default stand state are 0.645 m in length, 0.28 m in width, and 0.4 m in height, with a total weight of approximately 13 kg. Given its size and torque capabilities, it serves as an excellent platform for developing our dynamic locomotion strategy.

The base frame of the Go1 robot is located at the geometric center of the trunk. For simplicity, and due to the absence of external kinematic/dynamic libraries during training, such as Pinocchio [42] (which is utilized for deploying the policy on the real robot), we have assumed the origin of the base link placed with the CoM in this work.
All reference frames are aligned according to the convention that they align with the base frame when

the robot is in its fully stretched configuration (also known as the zero-configuration, where all joints are set to zero). In this setup, the X-axis (red) extends forward in the robot's direction (towards the head), the Y-axis (green) points laterally to the left side of the robot, and the Z-axis (blue) points upward, perpendicular to the trunk. In Fig. 2.1, the robot is illustrated with all its reference frames clearly represented.

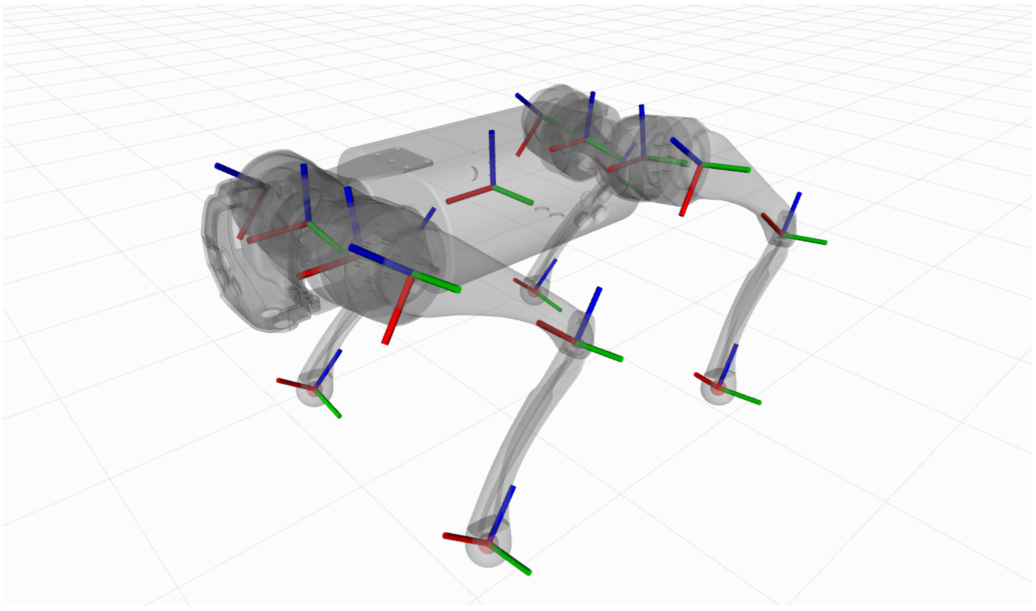


Figure 2.1: Standard definition of the link frames, using the RGB color convention for representing the X, Y, and Z axes

2.2 Robot Control

Legged robots, such as quadrupeds in our case, cannot fully describe their configuration solely through joint angles due to the free-floating nature of the base, which is not fixed relative to the world frame (inertial frame). To address this issue, an excess-coordinate method is employed, introducing a 6-DoFs virtual kinematic chain between the world frame W and the base frame B. This approach allows the legged robot to be represented similarly to a fixed-base manipulator. It provides a comprehensive description of the robot's configuration by extending the generalized coordinate vector to include both the base and joint configurations.

If Euler angles are chosen to parameterize the robot's orientation, the configuration of the robot is then described by the unactuated base coordinates $\mathbf{q}_b$, which represent the position and orientation of the base frame B with respect to W, and the actuated joint coordinates $\mathbf{q}_j$.
Consequently, the full generalized coordinate vector $\mathbf{q}$ is expressed as follows:

$$\mathbf{q} = \begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_j \end{bmatrix} \in \mathbb{R}^{n+6} \tag{2.1}$$

The pose of each link, including its position and orientation, can be derived from the base pose and joint configuration using the robot's Forward Kinematic (FK). To represent the orientation and angular velocity of the floating base, Euler angles and Euler rates can be adopted. However, this method may suffer from singularities, such as gimbal lock, which occur when two axes become aligned. An alternative representation is the use of quaternions, which offer greater numerical stability and do not suffer from singularities in their representation. By adopting quaternions, the $\mathbf{q}_b$ vector $\in \mathbb{R}^3 \times \mathbb{SO}^3$ where $\mathbb{SO}^3$ is the Special Orthogonal Group of order 3.

Given the generalized coordinate vector, the dynamics of the floating base quadruped robot can be expressed by the following equation:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{S}^T\boldsymbol{\tau} + \mathbf{J}_c(\mathbf{q})^T\mathbf{f} \tag{2.2}$$

where:

- $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{(n+6)\times(n+6)}$ is the inertia matrix
- $\mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) \in \mathbb{R}^{n+6}$ represents the Coriolis, centrifugal, and gravitational effects (bias forces)
- $\mathbf{S}^T \in \mathbb{R}^{(n+6)\times n}$ is the selection matrix responsible for mapping the actuated torques to the joints
- $\boldsymbol{\tau} \in \mathbb{R}^n$ is the vector of joint torques
- $\mathbf{J}_c(\mathbf{q}) \in \mathbb{R}^{6\times(n+6)}$ is the Jacobian matrix that maps joint velocities to contact velocities
- $\mathbf{f} \in \mathbb{R}^6$ denotes the external forces at the contact points (e.g., Ground Reaction Forces (GRFs) at the feet), in our case, we have point feet that do not exert moments so $\mathbf{f} \in \mathbb{R}^3$

The selection matrix $\mathbf{S}^T$ models the fact that the base variable is underactuated, ensuring that only the actuated joints are influenced by $\boldsymbol{\tau}$ and is defined as:

$$\mathbf{S}^T = \begin{bmatrix} \mathbf{0}_{6\times n} \\ \mathbf{I}_{n\times n} \end{bmatrix} \tag{2.3}$$

The Jacobian matrix $\mathbf{J}_c(\mathbf{q})$ is defined as following:

$$\begin{gathered} \mathbf{J}_c = \begin{bmatrix} \mathbf{J}_{cb} & \mathbf{J}_{cj} \end{bmatrix} \in \mathbb{R}^{6\times(n+6)} \\ \mathbf{J}_{cb} = \begin{bmatrix} \mathbf{I}_{3\times3} & -\left[\mathbf{x}_\mathrm{f} - \mathbf{x}_b\right]_\times \\ \mathbf{0} & \mathbf{I}_{3\times3} \end{bmatrix} \in \mathbb{R}^{6\times6} \quad \mathbf{J}_{cj} = \frac{\partial(\mathbf{x}_\mathrm{f} - \mathbf{x}_b)}{\partial\mathbf{q}} \quad \in \mathbb{R}^{6\times n} \end{gathered} \tag{2.4}$$

where $\mathbf{x}_\mathrm{f}$ is the position of the contact frame (foot) and $\left[\cdot\right]_\times$ is the skew-symmetric operator defined as:

$$\left[\mathbf{v}\right]_\times = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \tag{2.5}$$

The selection matrix indicates that it is possible to partition the dynamic model into unactuated and actuated components, which can be expressed in matrix form as follows:

$$\begin{bmatrix} \mathbf{M}_b(\mathbf{q}) & \mathbf{M}_{bj}(\mathbf{q}) \\ \mathbf{M}_{jb}(\mathbf{q}) & \mathbf{M}_j(\mathbf{q}) \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_b \\ \ddot{\mathbf{q}}_j \end{bmatrix} + \begin{bmatrix} \mathbf{h}_b(\mathbf{q}, \dot{\mathbf{q}}) \\ \mathbf{h}_j(\mathbf{q}, \dot{\mathbf{q}}) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{\tau}_j \end{bmatrix} + \begin{bmatrix} \mathbf{J}_{cb}^T(\mathbf{q})\mathbf{f} \\ \mathbf{J}_{cj}^T(\mathbf{q})\mathbf{f} \end{bmatrix} \tag{2.6}$$

where $\mathbf{M}_b(\mathbf{q})$, $\mathbf{M}_{bj}(\mathbf{q})$, $\mathbf{M}_{jb}(\mathbf{q})$, and $\mathbf{M}_j(\mathbf{q})$ denote the subcomponent of the inertia matrix, and $\mathbf{h}_b$,$\mathbf{h}_j$ represent the nonlinear effects (Coriolis, centrifugal, and gravitational) for the base and joints, respectively. This decomposition makes it more evident that the motion of the base is not directly controllable, rather, it can only be influenced through the forces applied at the contact points.

2.2.1 Trajectory tracking and Gravity Compensation

To track a given CoM (CoM) trajectory, a method for controlling the robot's pose is required. In this work, we employ a kinematics-based approach combined with gravity compensation, implemented via a Whole Body Control (WBC). This choice was made for simplicity and efficiency, given the assumption that all four legs maintain ground contact throughout the trajectory tracking process. Additionally, we assume that the desired CoM pose remains within the feasible set of configurations achievable from the initial standing posture of the robot.
Once the desired CoM trajectory is specified, IK is employed to compute the joint configurations and velocities that will achieve this trajectory. The computed joint positions $\mathbf{q}^d$ and velocities $\dot{\mathbf{q}}^d$ are then passed to a Proportional-Derivative (PD) controller, along with a feedforward torque $\boldsymbol{\tau}_{ff}$ derived from gravity compensation.
Given a desired CoM pose ${}_w\mathbf{x}_c^d$ (that due to our assumption is coincident with the base frame becoming ${}_w\mathbf{x}_b^d$)in the world frame W, we need to compute the desired foot positions relative to the base frame B.
This transformation is achieved through:

$$ {}_b\mathbf{x}_{f_i}^d = ({}_w\mathbf{R}_b^d)^{-1}({}_w\mathbf{x}_{\mathrm{f}_i} - {}_w\mathbf{x}_b^d) \tag{2.7}$$

where ${}_b\mathbf{x}_{f_i}$ is the position of the i-th stance foot in the world frame, ${}_w\mathbf{x}_b^d$ is the desired base position in the world frame, and ${}_w\mathbf{R}_b^d$ is the rotation matrix representing the orientation of the desired base frame with respect to the world frame.
Using the desired foot positions ${}_b\mathbf{x}_{f_i}^d$ expressed in the base frame, we compute the desired joint angles $\mathbf{q}_i^d$ for each leg using IK, which can be solved either analytically or numerically depending on the complexity of the leg kinematics.
The process can be summarized as:

$$\mathbf{q}_i^d = \mathbf{IK}({}_b\mathbf{x}_{f_i}^d) \tag{2.8}$$

where $\mathbf{IK}$ represents the inverse kinematics solver.

Instead of using a simple finite difference to compute the desired joint velocities $\dot{\mathbf{q}}^d$, which may introduce numerical errors and instability, a more robust method involves the usage of differential kinematics thought the relationship between joint velocities and the end-effector velocity:

$$\dot{\mathbf{q}}_i^d = \mathbf{J}_{cj,i}^{\dagger}(\mathbf{q}_i^d)\,{}_b\dot{\mathbf{x}}_{f_i}^d \tag{2.9}$$

where $\mathbf{J}_{cj,i}$ is the i-th leg Jacobian matrix, $[\cdot]^{\dagger}$ is the pseudo-inverse operator[1], and ${}_b\dot{\mathbf{x}}_{f_i}^d$ is the i-th desired end-effector (foot) velocity in the base frame. Unless specified henceforth, the vectors are assumed to be specified in the world frame W.

To maintain stability and improve the tracking of the desired CoM trajectory, gravity compensation is applied via a feedforward torque. The relationship between the contact forces $\mathbf{f} \in \mathbb{R}^n$ and the wrench $\mathbf{w} \in \mathbb{R}^6$ at the base is given by:

$$\mathbf{J}_{cb}^T\mathbf{f} = \mathbf{w} \tag{2.10}$$

where $\mathbf{J}_{cb}^T$ maps contact forces into the wrench acting on the base. Since we assume that all four feet remain in contact with the ground, the contact wrench matrix is defined as:

$$\mathbf{J}_{cb}^T = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{I}_{3\times3} & \mathbf{I}_{3\times3} & \mathbf{I}_{3\times3} \\ [\mathbf{x}_{\mathrm{f,FL}} - \mathbf{x}_b]_\times & [\mathbf{x}_{\mathrm{f,FR}} - \mathbf{x}_b]_\times & [\mathbf{x}_{\mathrm{f,RL}} - \mathbf{x}_b]_\times & [\mathbf{x}_{\mathrm{f,RR}} - \mathbf{x}_b]_\times \end{bmatrix} \tag{2.11}$$

where $\mathbf{x}_{\mathrm{f,FL}}, \mathbf{x}_{\mathrm{f,FR}}, \mathbf{x}_{\mathrm{f,RL}}, \mathbf{x}_{\mathrm{f,RR}}$ denote the positions of the front-left, front-right, rear-left, and rear-right feet with respect to the world frame, and $\mathbf{x}_b$ is the base position.

[1]In our case is a pure inversion because the leg is nonredundant to control only the position of the foot

To compensate for gravity, the desired wrench $\mathbf{w}^d$ to be rendered at the base for compensating gravity is modeled as:

$$\mathbf{w}^d = \begin{bmatrix} 0 \\ 0 \\ mg \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.12}$$

where m represents the total mass of the robot and $g = 9.81 \left[\frac{m}{s^2}\right]$ is the gravitational acceleration. By solving the equation 2.10, we obtain the desired contact forces:

$$\mathbf{f}^d = (\mathbf{J}_{cb}^T)^{\dagger}\mathbf{w}^d \tag{2.13}$$

These contact forces are then used to compute the vector of feed-forward torques $\boldsymbol{\tau}_{ff} \in \mathbb{R}^n$ by simple mapping with the joint Jacobian:

$$\boldsymbol{\tau}_{ff} = -\mathbf{J}_{cj}^T\mathbf{f}^d = -\mathbf{J}_{cj}^T(\mathbf{J}_{cb}^T)^{\dagger}\mathbf{w}^d \tag{2.14}$$

where $\mathbf{J}_{cj}^T$ maps the contact forces to the joint torques, and the negative sign accounts for the fact that contact forces are forces that the environment exerts on the robot.

With the desired joint configurations $\mathbf{q}^d$, velocities $\dot{\mathbf{q}}^d$, and feedforward torque $\boldsymbol{\tau}_{ff}$ computed, the final torque command $\boldsymbol{\tau}$ is generated using a Proportional-Derivative (PD) controller:

$$\boldsymbol{\tau} = \mathbf{K}_p(\mathbf{q}^d - \mathbf{q}) + \mathbf{K}_d(\dot{\mathbf{q}}^d - \dot{\mathbf{q}}) + \boldsymbol{\tau}_{\boldsymbol{ff}} \tag{2.15}$$

where $\mathbf{K}_p \in \mathbb{R}^{n\times n}$ and $\mathbf{K}_d \in \mathbb{R}^{n\times n}$ are the proportional (stiffness) and derivative (damping) gains, respectively.
This control law ensures that the robot accurately tracks the desired joint trajectories while compensating for gravity, leading to effective CoM trajectory tracking during motion.

2.3 Reinforcement Learning in Robotics

Recently, with the impressive advancements in simulation technologies and computational power, RL has emerged as a powerful solution for developing sophisticated control strategies in robotics. These advancements have allowed robots to learn and perform complex tasks, making possible something that would have been unimaginable just a few years ago. Numerous research works have showcased the groundbreaking potential of RL in various robotic applications, including locomotion, manipulation, and navigation tasks [43, 4, 44, 45].
In this section, we will introduce some fundamental concepts of RL and provide a brief overview of one of the most widely used RL algorithms in robotics today: Proximal Policy Optimization (PPO) [46]. We will delve into the core ideas and mathematical principles that make PPO a leading choice for RL-based robotic control strategies.

2.3.1 Key Concepts of Reinforcement Learning

RL, a subfield of machine learning, focuses on the study of agents that learn optimal behaviors through trial and error, interacting with their environment to maximize cumulative rewards. This approach allows robots to adapt to a wide range of dynamic and uncertain scenarios, making it an invaluable tool in robotic control.

Let's explore in more detail the fundamental components of RL [47]:

- **Agent**: The agent is the central entity that learns and makes decisions through interaction with the environment. In our context, this typically refers to the robot or a specific part of the robot interacting with its surroundings.
- **Environment**: The environment represents the world in which the agent resides and interacts, providing feedback in response to the agent's actions. This environment can be either simulated or real.
- **State** (s): The state is a complete or partial representation of the environment at a specific moment in time. This information is crucial for the agent to make informed decisions.
- **Action** (a): The action represents the decision made by the agent based on its current state. Actions can belong to either a discrete or continuous space, influencing which algorithms can be applied.
- **Reward** (r): The reward is a quantitative measure provided as feedback by the environment, indicating how well or poorly the agent's action contributed to achieving the desired goal. The reward function R depends on the current state, the action taken, and the subsequent state reached after performing the action:

  $$r_t = R(s_t, a_t, s_{t+1})$$
- **Policy** (π): The policy defines the agent's behavior by mapping states to actions. It can be deterministic, $\mu(s_t)$, or, as in our case, stochastic, $\pi(\cdot|s_t)$. In Deep Reinforcement Learning (Deep-RL), where policies are represented by Neural Network (NN), the policy depends on the network parameters, often denoted by θ or ϕ. Thus, the stochastic policy is expressed as:

  $$a_t \sim \pi_\theta(\cdot|s_t)$$
- **Trajectory** (τ): A trajectory is the entire sequence of states and actions recorded during the agent's interaction with the environment:

  $$\tau = (s_0, a_0, s_1, a_1, \ldots)$$

The ultimate goal of RL is to find an optimal policy π^* that maximizes the expected return when the agent acts according to this policy. For a stochastic policy and environment transitions, the expected return, denoted by $J(\pi)$, is defined as:

$$J(\pi) = \underset{\tau \sim \pi}{E}[R(\tau)]$$

Therefore, the final objective of RL can be expressed as:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.16}$$

The method by which this objective is achieved is the key factor that differentiates various RL algorithms. One of the most significant distinctions in RL algorithms is whether they incorporate a model of the environment to which the agent has access. An environment model refers to a function capable of predicting state transitions based on the current state and the action performed by the agent. Having such a model allows the agent to anticipate outcomes by exploring potential future scenarios, thereby significantly enhancing the sample efficiency of the learning process.

Unfortunately, in most cases, a model of the environment is not readily available to the agent, meaning the agent must learn it purely through interaction experience. This limitation has a substantial

impact on sample efficiency and increases the risk of suboptimal performance when the agent is deployed in a real-world environment. Despite this drawback, model-free algorithms are generally easier to implement and fine-tune, which is why they currently remain the most popular and widely adopted approach in RL, especially in the robotic control field.
Another important distinction between RL algorithms lies in how they interact with the policy during the training process:

- **On-Policy Algorithms**: These algorithms learn about the policy while actively following it. In other words, the policy used to generate actions is the same as the one being improved upon during training. The PPO algorithm, which we will discuss later, belongs to this category. On-policy algorithms tend to be more stable during the learning process but require a large number of samples, as each learning step necessitates fresh data collected by following the current policy. Therefore, parallelization plays a crucial role in enhancing the efficiency of this class of algorithms.

- **Off-Policy Algorithms**: These algorithms learn about the policy from data generated by a different policy. Off-policy methods can reuse past experiences stored in a replay buffer, making them more sample-efficient. However, despite this advantage, they can be more challenging to stabilize and often require significant memory resources to store a sufficiently large quantity of historical data.

In summary, these fundamental concepts form the basis of RL, where the agent learns by interacting with its environment. Fig. 2.2 illustrates this interaction schema, showing how the Go1 robot (agent) engages with the Isaac Sim environment, generating actions, receiving rewards, and observing state changes as it iteratively refines its policy.

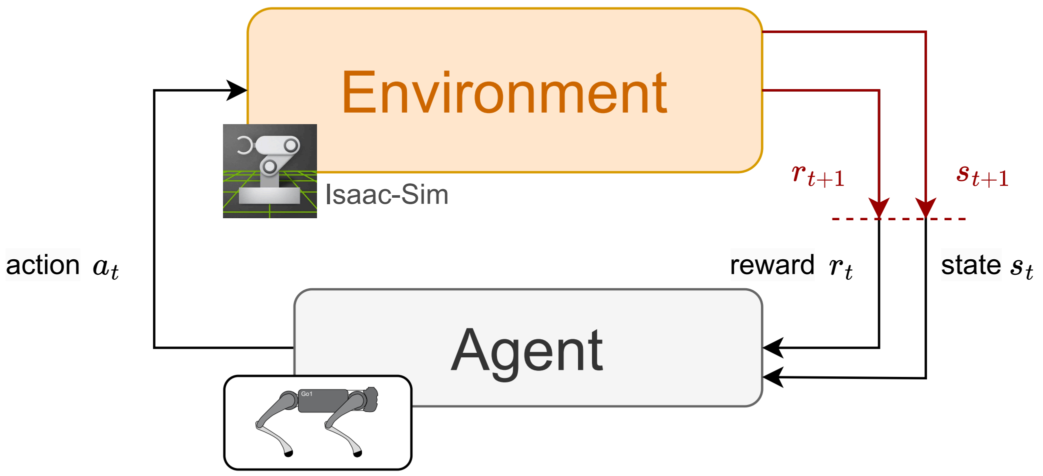


Figure 2.2: Interaction schema between the Go1 robot (agent) and Isaac Sim (environment), showing the exchange of actions, state, and rewards between these two entities

2.3.2 Proximal Policy Optimization (PPO)

As previously mentioned, PPO is one of the most popular on-policy RL algorithms, widely adopted for both its balance of performance and implementation simplicity. This algorithm is versatile, being applicable to environments with either discrete or continuous action spaces. PPO is inspired by the Trust Region Policy Optimization (TRPO) algorithm [48], where the core idea is to limit the policy update at each training epoch, thereby preventing performance collapse due to overly large steps toward potentially misleading optimal paths.

A key improvement of PPO over TRPO lies in its mathematical formulation, transitioning from a second-order method to a first-order method while maintaining comparable performance.

There are two versions of the PPO algorithm: PPO-Penalty, which, like TRPO, incorporates the Kullback–Leibler (KL) divergence, and PPO-Clip, which omits the KL divergence term and instead uses a clipped surrogate objective function. Although these two versions are formally defined differently, they tend to perform similarly. For this reason, we will focus on the simpler PPO-Clip method in our discussion.

Let's begin by defining the policy update step as:

$$\theta_{k+1} = \arg\max_{\theta} L^{\text{CLIP}}(\theta) \tag{2.17}$$

Before introducing $L^{\text{CLIP}}(\theta)$, we first define the probability ratio between the new policy and the old policy:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \tag{2.18}$$

This ratio measures the divergence between the two policies: if $r_t(\theta) > 1$, the action a_t at state s_t is more likely under the current policy, while if $0 \leq r_t(\theta) \leq 1$, the action is more likely under the old policy.

The objective function $L^{\text{CLIP}}(\theta)$ is then defined as:

$$L^{\text{CLIP}}(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{2.19}$$

where:

- $\hat{E}_t\left[\cdot\right]$ is the expectation operator, representing the empirical average over a finite batch of samples.
- $\hat{A}_t$ is an estimator of the advantage function at timestep t.
- ϵ is a hyperparameter (e.g., $\epsilon = 0.2$) that defines the allowable deviation between the new and old policies.

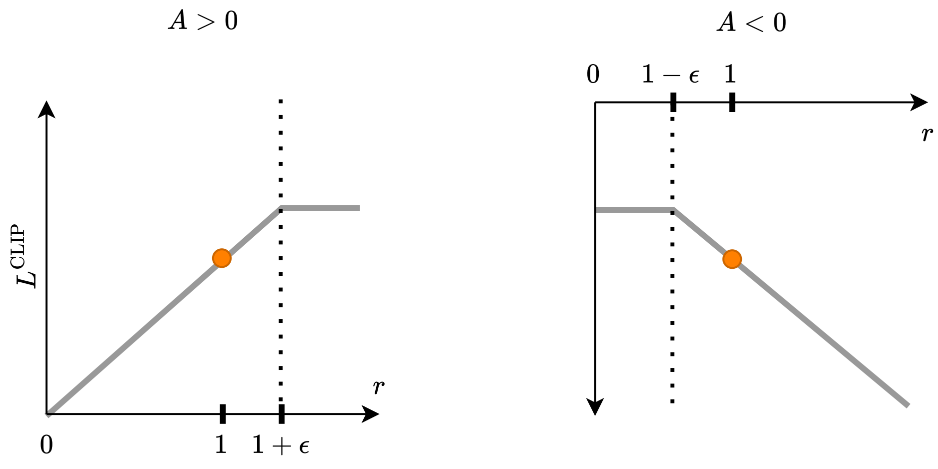


Figure 2.3: Surrogate function L^{CLIP} as a function of the probability ratio r for a single time step. The orange dot represent the starting point for the optimization i.e. $r = 1$

The clipping operation serves as a regularization mechanism, preventing excessive changes to the policy and ensuring stability (see Fig. 2.3). One of the significant advantages of this approach is that the policy update step can be performed using stochastic gradient ascent with standard optimizers such as Adam [49], which substantially reduces computational complexity. Although occasional suboptimal decisions may occur, the algorithm achieves an excellent balance between optimization speed and stability.

2.4 Simulation Solutions for Accelerating Locomotion Learning

Developing new control algorithms and researching novel RL solutions directly on real-world hardware is often impractical due to constraints related to time, cost, and safety. Simulators play an essential role in robotics by providing a safe, cost-effective, and controlled environment for prototyping these developments. The field of robotics simulation is diverse, offering various solutions each one with its strength point, differing in terms of physics accuracy, computational efficiency, photorealism, and integration with external libraries and frameworks. Among the most popular simulators are Gazebo, PyBullet, MuJoCo, and more recently, NVIDIA's Isaac Sim.

For this work, Isaac Sim was selected due to its impressive capabilities in terms of parallelization, particularly in the context of RL-based locomotion, as demonstrated in recent studies such as in the work of Rudinti et al. [50]. Isaac Sim [51] is a high-fidelity simulation platform developed by NVIDIA, designed specifically for AI applications in robotics and autonomous machines.
Built on NVIDIA's Omniverse platform, Isaac Sim leverages the GPU-accelerated PhysX physics engine, providing realistic simulation with accurate rigid-body dynamics, soft-body physics, and complex environmental interactions.

To further enhance the capabilities of Isaac Sim, NVIDIA Orbit (recently renamed to Isaac-Lab) was developed in collaboration with ETH Zurich and the University of Toronto [52]. Orbit is a unified and modular framework tailored for robotics and robot learning. It includes pre-configured models of quadruped robots from various companies, such as Unitree and ANYbotics, and offers seamless integration with state-of-the-art RL libraries like Stable Baselines [53] and RSL-RL [54]. One of Orbit's key strengths is its structured RL tasks with modular components, which significantly accelerate the development process and provide a robust foundation upon which researchers can expand and adapt the framework to meet their specific needs.

Isaac Sim's capacity for massive parallelization is fully leveraged by the vectorized environments managed through the Orbit framework, which seamlessly integrates with the previously mentioned RL libraries. This parallelization capability allows for the rapid simulation of thousands of trials concurrently, dramatically reducing training time. As a result, it effectively mitigates the challenges associated with the millions of training steps typically required for policy training, enabling more efficient development of robust RL solutions.

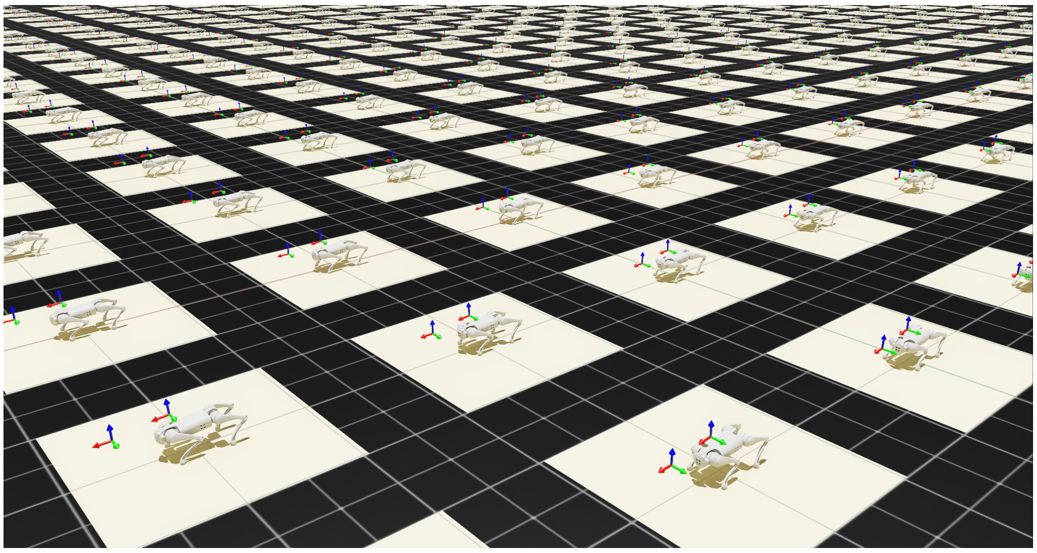


Figure 2.4: Showcase of the parallelization capabilities of Isaac Sim with the Orbit framework

Orbit is designed to request and apply a new action at each timestep. However, in contrast to this, our system is based on the parametrization of a trajectory. This trajectory parametrization is derived from a single policy action, which is requested only once at the beginning of the episode and then applied throughout the entire episode. This abstraction of high-level planning necessitated modifications to how Orbit executes its RL tasks.
Another distinction lies in the need to differentiate between the computation of a running reward and the computation of a reward that is provided only at the conclusion of the entire episode. Additionally, our approach diverges from Orbit's traditional use case in that the robot cannot be reset mid-episode in the event of failure, which enforces synchronous parallelization.
In standard tasks, when a robot fails, it is reset. This is feasible because the task is driven by a sequence of actions, with each action being derived from the robot's current state. For example, if the task involves following a desired velocity and the robot falls, the system can reset only the failed robot while allowing the others to continue. Since actions and rewards are computed independently at each timestep, there is no need to suspend the failed execution or allow it to continue in a compromised state.
In our case, however, the entire episode depends on the action computed at the start of the episode, making it impossible to reset a robot and request a new action mid-episode. We must wait for the episode to complete before resetting all robots and requesting a new action for the next episode.

The Alg. 1 provided above illustrates our implementation of the RL planning task for single-action episodes. This contribution extends Orbit's capabilities, enabling the creation of RL tasks that involve high-level planning, where a single action governs the entire episode and different reward structures, such as those in our system, can be employed.

Algorithm 1 Single action episode RL

```
1:  function STEP(action):
2:
3:      action_manager.process_action(action)        ▷ Process actions, obtaining the trajectory
4:      running_reward ← 0
5:
6:      for $i$ ← 0 to max_episode_length do
7:          action_manager.apply_action()
8:          sim.step()
9:          running_reward += running_reward_manager.compute()
10:         event_manager.apply(mode = "interval")           ▷ Execute interval events
11:     end for
12:
13:     final_reward ← final_reward_manager.compute()
14:     total_reward ← compute_total_reward(running_reward, final_reward)
15:
16:     scene_manager.reset("all")
17:
18:     command_manager.compute()         ▷ Update command and compute observations
19:     obs_buf ← observation_manager.compute()
20: end function
```

By "process action," we refer to all preprocessing operations required for the action to be applied at each time step. In our case, since the action corresponds to the CoM trajectory parameters, the trajectory is computed during the preprocessing phase. During action application, the trajectory is evaluated, and inverse kinematics (IK) is used for tracking. Additionally, some periodic events may be necessary for checks or to dynamically adjust certain characteristics of the environment. These events are evaluated and applied at the end of each time step.

2.4.1 Sim-to-Real transfer

Deploying a trained policy to real hardware presents several challenges. Even when the policy does not require further fine-tuning (typically in locomotion tasks where joint or torque references are generated, and where issues are addressed through techniques such as randomization of mass, CoM displacement, motor maximum torque, and other property randomization) specific interfacing with the robot remains necessary.
Implementing the policy on a real robot is a complex task that depends on the robot's control strategy. Furthermore, deploying the saved model is influenced by the particular RL library used. For example, there may be instances where the model needs to be integrated into a C++ control system, despite being accessible only through Python library. Since no RL framework offers direct solutions for such integrations, ad-hoc methods are required to accomplish this deployment process.

A common approach for deploying the RL policy on the Go1 robot, following training in various locomotion learning frameworks that have been progressively developed up to Isaac-Lab, is to use the sim-to-real solution introduced by Margolis et al. [55].
In contrast, we utilize Locosim [56], a robotic framework primarily developed by Michele Focchi. Locosim is a didactic framework designed for learning and testing basic control schemes on both quadruped robots, and manipulators. It offers a comprehensive environment for simulating and experimenting with robotic controllers, providing an interface that facilitates the implementation and evaluation of various control strategies. The framework is composed of a ROS-based control architecture, with a C++ roscontrol node that interfaces with a Python ROS node, enabling seamless integration with the Gazebo simulator. Locosim has been successfully tested on different real robotic platforms, including the UR5, Aliengo, and Go1 robots, and serves as an accessible and practical tool

for researchers and practitioners aiming to develop and validate their control strategies on real robotic hardware.

The RSL RL framework facilitates the straightforward export of trained policies in the widely-used ONNX format [57]. Once the model is saved in this format, it can be loaded and actions inferred using the ONNXRuntime Python library. The same control strategy applied in Locosim is also used in Orbit, ensuring consistency across simulation and real-world deployment.

2.5 Jump Taxonomy

Although jumping is an inherent behavior across the animal kingdom, it constitutes a complex and dynamic movement, presenting significant challenges for its replication in robotic systems. Different species have evolved distinct jumping techniques and capabilities. However, from a biomechanical perspective, the jumping action can be identified in three main phases: thrusting, flying, and landing (see Fig. 2.5).
From a physics standpoint, during the airborne phase, no additional thrust can be generated, meaning the jump trajectory is solely determined by the conservation of momentum obtained at lift-off. Consequently, the thrusting phase is crucial for the overall outcome of the jump, as it involves the force generation necessary to build up the need momentum at lift-off necessary to reach the desired target. Once the legs leave the ground, the trajectory of the jump follows the principles of ballistic motion, with gravity being the only acting force on the system. For this reason, the jump trajectory lies in the vertical plane that connect the starting location to the target location
In the following sections, we will provide a more detailed analysis of each phase of the jump.

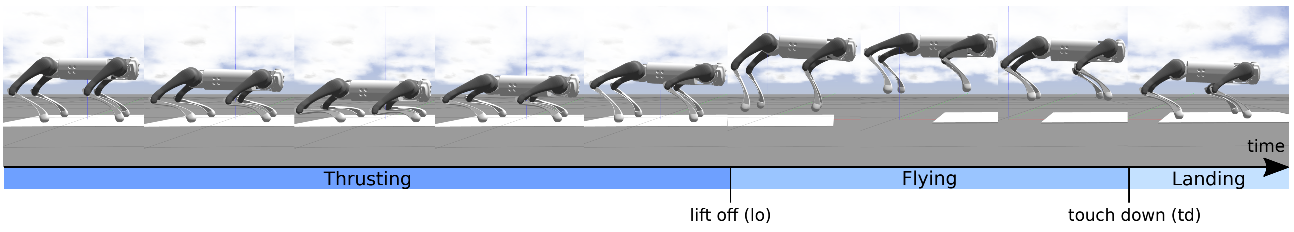


Figure 2.5: Robot executing the three stages of a jump (in Locosim)

2.5.1 Thrust Phase

As previously discussed, the thrust phase is the primary phase that determines the jump's outcome. Even minor variations during this phase can compromise the overall trajectory, potentially leading to failure or preventing the attainment of the desired target. Drawing insights from nature and biomechanics, we can state that the thrust phase involves an initial compression phase, which is immediately followed by an explosive decompression stage, where the system is pushed to its mechanical limits. This process can be likened to a spring's compression and subsequent release. In the context of quadruped jumping, this behavior is essential as it allows the robot to exploit the full range of its joints to generate maximum acceleration and force.
Precise timing and effective energy management are critical for achieving the optimal balance between acceleration, flight duration, jump apex, and potential aerial maneuvers. Selecting the appropriate orientation and positioning throughout the thrust phase is essential for distributing force efficiently, preventing imbalances, and avoiding slippage. This entire phase demands the generation of maximum torque while maintaining joint velocity and position within safe limits, considering constraints imposed by the terrain, such as frictional resistance.

2.5.2 Flight Phase

The flight phase begins as soon as the robot loses contact with the ground and becomes airborne, with its motion governed solely by the laws of projectile motion, as determined by the lift-off configuration. Since this phase is purely the result of momentum conservation, it does not actively contribute to reaching the desired target. However, during this stage, minor adjustments to leg positioning, and consequently the robot's orientation, may occur to optimize obstacle avoidance and enhance landing posture. The primary mechanisms involved include leg retraction, followed by a sequential extension after reaching the apex, which helps cushion the impact during touchdown and mitigate landing forces. The degree to which the legs are extended significantly affects the robot's stiffness upon landing; less extended legs result in a softer, less stiff landing, while more extended leg configurations provide a greater joint range for deceleration.

2.5.3 Landing Phase

The landing phase involves re-establishing ground contact and absorbing impact forces while maintaining the robot's balance and preventing structural damage. The orientation and leg extension set during the flight phase play a crucial role in optimizing impact absorption at this stage. Upon touchdown, the robot should behave like a compliant spring, compressing to absorb the impact forces and gradually returning to its default standing configuration. This process also dissipates kinetic energy, particularly if there is any horizontal velocity, which helps prevent the robot from tipping over. This sequential adjustment is essential for ensuring stability and preventing excessive stress on the robot's structure.

An important contribution to managing the last two phases was made by Roscia et al. [5]. They developed an optimization-based reactive Landing Controller (LC) that relies on proprioceptive measures to manage the flying and landing phases. Using a Variable Height Springy Inverted Pendulum (VHSIP) model, the controller continuously adjusts the robot's feet positions based on the horizontal velocity, enabling stable landings even at horizontal speeds up to 3 m/s. This approach minimizes bouncing, slippage, and improper contact of the trunk with the ground, providing a solid foundation for more advanced control strategies.
In this work, a LC is omitted during the training phase to maintain a simplified learning approach. However, the presence of a LC has a significant impact on jump performance, as will be demonstrated in the results section. In future work, we plan to design a dedicated NN to manage the landing phase and adjust the robot's posture during the aerial phase.

2.6 Balistic Problem

We have analyzed the taxonomy of jumping and identified the thrust phase as the most critical component, as it determines the overall jump outcome. Given that the jump is governed by the conservation of momentum and that, once airborne, the system follows a ballistic trajectory, the landing position is solely dependent on the lift-off configuration. Therefore, it is essential to understand how to select this configuration appropriately to reach the desired target. In the following section, we will examine the relationship between the lift-off configuration and the landing position, highlighting why jump planning is a highly complex and non-trivial task.

We define the CoM lift-off configuration as the pair of vectors $(\mathbf{s}_{lo}, \dot{\mathbf{s}}_{lo})$, which represent the CoM lift-off position $\mathbf{c}_{lo}$ and trunk orientation $\mathbf{\Phi}_{lo}$ expressed in Euler angles, as well as the corresponding CoM lift-off linear $\dot{\mathbf{c}}_{lo}$ and angular velocities $\dot{\mathbf{\Phi}}_{lo}$.

Additionally, we define the CoM target configuration $\mathbf{s}_{tg}$ as the vector containing the desired CoM landing position $\mathbf{c}_{tg}$ and orientation $\mathbf{\Phi}_{tg}$.

$$\mathbf{s}_{lo} = \begin{bmatrix} \mathbf{c}_{lo} \\ \mathbf{\Phi}_{lo} \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \psi \\ \theta \\ \varphi \end{bmatrix} \qquad \dot{\mathbf{s}}_{lo} = \begin{bmatrix} \dot{\mathbf{c}}_{lo} \\ \dot{\mathbf{\Phi}}_{lo} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \\ \dot{\theta} \\ \dot{\varphi} \end{bmatrix} \qquad \mathbf{s}_{tg} = \begin{bmatrix} \mathbf{c}_{tg} \\ \mathbf{\Phi}_{tg} \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \psi \\ \theta \\ \varphi \end{bmatrix} \tag{2.20}$$

The projectile motion of the system can be described by the following set of equations where only the linear parts of $\mathbf{s}_{lo}$,$\dot{\mathbf{s}}_{lo}$, and $\mathbf{s}_{tg}$ are considered:

$$\begin{cases} \mathbf{c}_{tg,x} = \mathbf{c}_{lo,x} + \dot{\mathbf{c}}_{lo,x} T_{fl} \\ \mathbf{c}_{tg,y} = \mathbf{c}_{lo,y} + \dot{\mathbf{c}}_{lo,y} T_{fl} \\ \mathbf{c}_{tg,z} = \mathbf{c}_{lo,z} + \dot{\mathbf{c}}_{lo,z} T_{fl} - \frac{1}{2} g T_{fl}^2 \end{cases} \tag{2.21}$$

where T_{fl} represents the time of flight, and g denotes the acceleration due to gravity.

As a simplified case study, we consider the execution of a forward jump neglecting the orientation. Since the entire jump trajectory lies within the plane connecting the starting location to the target, we can analyze the ballistic motion within a two-dimensional side-view space (see Fig. 2.6), involving the z-axis and x-axis. Given the desired target position, and assuming for simplicity that the lift-off position is fixed, we aim to investigate the impact of the lift-off linear velocity on the jump outcome. The details of this simplified problem are outlined below:

$$\mathbf{c}_{lo} = \begin{bmatrix} x \\ z \end{bmatrix} \qquad \dot{\mathbf{c}}_{lo} = \begin{bmatrix} \dot{x} \\ \dot{z} \end{bmatrix} \qquad \mathbf{c}_{tg} = \begin{bmatrix} x \\ z \end{bmatrix}$$

$$\begin{cases} \mathbf{c}_{tg,x} = \mathbf{c}_{lo,x} + \dot{\mathbf{c}}_{lo,x} T_{fl} \\ \mathbf{c}_{tg,z} = \mathbf{c}_{lo,z} + \dot{\mathbf{c}}_{lo,z} T_{fl} - \frac{1}{2} g T_{fl}^2 \end{cases} \tag{2.22}$$

To eliminate the artificial constraint of a fixed flight time, we first express it explicitly and then substitute the obtained value into the second equation:

$$\begin{cases} T_{fl} = \dfrac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} \\ \mathbf{c}_{tg,z} = \mathbf{c}_{lo,z} + \dot{\mathbf{c}}_{lo,z} \dfrac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} - \dfrac{1}{2} g \left(\dfrac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} \right)^2 \end{cases} \tag{2.23}$$

We can now take the resulting equation and express $\dot{c}_{lo,z}$ as a function of the remaining terms:

$$\begin{aligned} \mathbf{c}_{tg,z} &= \mathbf{c}_{lo,z} + \dot{\mathbf{c}}_{lo,z} \frac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} - \frac{1}{2} g \left(\frac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} \right)^2 \\ \dot{\mathbf{c}}_{lo,z} \frac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} &= -\mathbf{c}_{lo,z} + \frac{1}{2} g \left(\frac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} \right)^2 + \mathbf{c}_{tg,z} \\ \dot{\mathbf{c}}_{lo,z} &= -\mathbf{c}_{lo,z} \frac{\dot{\mathbf{c}}_{lo,x}}{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}} + \mathbf{c}_{tg,z} \frac{\dot{\mathbf{c}}_{lo,x}}{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}} + \frac{1}{2} g \frac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} \\ &= \frac{\mathbf{c}_{tg,z} - \mathbf{c}_{lo,z}}{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}} \dot{\mathbf{c}}_{lo,x} + \frac{1}{2} g \left(\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x} \right) \frac{1}{\dot{\mathbf{c}}_{lo,x}} \end{aligned} \tag{2.24}$$

Combining all the elements, we obtain:

$$\begin{cases} T_{fl} = \dfrac{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}}{\dot{\mathbf{c}}_{lo,x}} \\ \dot{\mathbf{c}}_{lo,z} = \dfrac{\mathbf{c}_{tg,z} - \mathbf{c}_{lo,z}}{\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}} \dot{\mathbf{c}}_{lo,x} + \dfrac{1}{2} g \left(\mathbf{c}_{tg,x} - \mathbf{c}_{lo,x}\right) \dfrac{1}{\dot{\mathbf{c}}_{lo,x}} \end{cases} \tag{2.25}$$

From Eq. 2.25, we observe that the relationship between $\dot{\mathbf{c}}_{lo,z}$ and $\dot{\mathbf{c}}_{lo,x}$ forms a hyperbolic function of the form $y = ax + \frac{b}{x}$.
Additionally, the flight time is inversely proportional to the horizontal component of the lift-off velocity. This observation aligns with intuition: if the horizontal component is small, a larger vertical component is required to reach the target. Consequently, a higher vertical component results in an increased flight time compared to a more balanced velocity distribution.

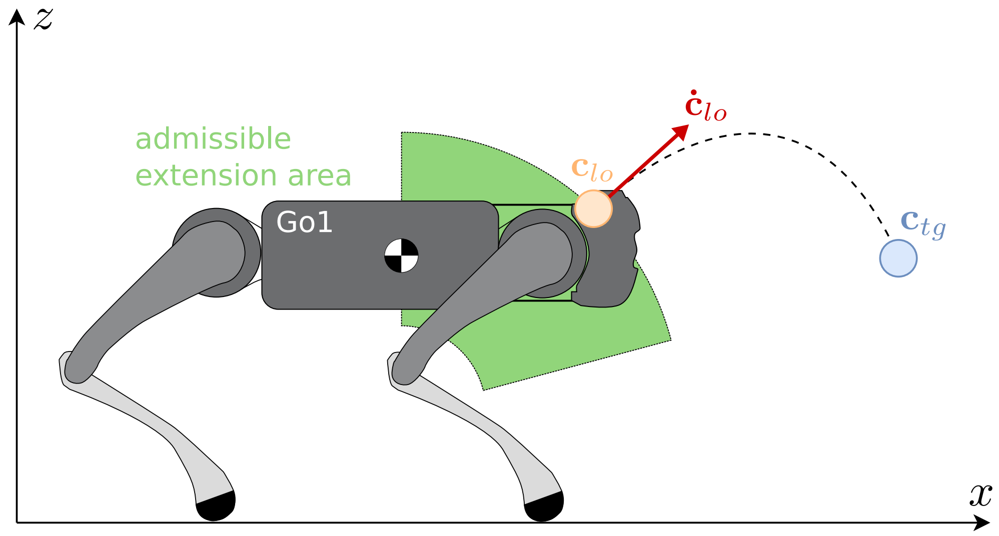


Figure 2.6: Side-view of the ballistic problem

For instance, if we aim to find a solution that minimizes the flight time we need to have a high horizontal velocity. This optimization problem must also consider the limitations of the robot system, such as the maximum achievable velocity that can limit the maximum vertical and horizontal velocity. This requires incorporating the robot's dynamics into our system model, along with a set of constraints such as joint position, velocity, and torque limits, which significantly increase the complexity of the problem and restrict the feasible range of lift-off velocities.
If we further relax the assumption of a fixed lift-off position and include it as a decision variable, while also constraining it to lie within the admissible extension area of the robot with all four legs in contact, the dimensionality of the optimization problem increases substantially. This results in a strongly non-convex optimization problem typically addressed using Nonlinear Programming (NLP) solver techniques.

This problem is often tackled in conjunction with trajectory planning during the thrust phase. It becomes evident why such a problem is inherently complex, and why artificial constraints, such as fixed thrust or flight times, are commonly employed alongside simplified robot models [7, 8]. These practices help reduce the number of variables to optimize, thereby decreasing computational complexity and time at the cost of flexibility. Moreover, NLP solvers are sensitive to initialization and are prone to getting trapped in local minima.
For these reasons, this approach is currently not suitable for addressing our jumping problem efficiently and in RT. Instead, a more efficient solution is needed, which is where GRL becomes advantageous.

In this context, the lift-off configuration serves as the primary action of our policy, as it is the crucial factor determining the overall jump outcome. Our approach involves the parameterization of the thrust trajectory, where the policy's action is determined only once before the jump begins and is then used to define the parameterized trajectory. This method is highly efficient, and since it is computed through a simple NN architecture, it operates with exceptional speed, enabling RT performance.
At each step of the control loop, a low-level controller tracks the desired trajectory, ensuring robustness and stability throughout the jump.

2.7 Computing the Thrust Trajectory

We have introduced the ballistic problem but have yet to address how to reach the desired lift-off configuration during the thrust phase. As previously discussed from biomechanical principles, this phase resembles a spring-like mechanism, characterized by an initial compression stage followed by an explosive decompression stage to generate the momentum necessary for lift-off.
Our objective is to model this behavior in the simplest yet most effective manner by leveraging engineering principles, rather than relying solely on an E2E approach. End-to-end (E2E) approaches learn the entire control policy directly from raw sensory inputs to action outputs, without the need for manually designed intermediate steps or features. This allows the policy to autonomously discover the optimal behavior but often requires significant training data and computational resources.

The most efficient approach is to parameterize the CoM trajectory in Cartesian space and then utilize IK to track it. This method ensures generality and independence from the robot's specific morphology. Once the trajectory is parameterized, RL can be employed to learn the trajectory parameters, which leads to a reduction in problem complexity and exploration requirements, while also enhancing feasibility checks and safety.

Among the various possible curve parameterizations, such as Splines and Hermite Curves, we have chosen to adopt the Bézier curve to parameterize the thrust trajectory. This choice is primarily motivated by the following characteristics: its simple yet efficient definition and computation, the fact that the entire curve is contained within the convex hull formed by its control points, and its advantageous analytical properties, particularly in derivative calculation. We will now provide a detailed explanation of the Bézier curve and its application in the parameterization of the thrust phase.

2.7.1 Bézier Curve

Named after the French engineer Pierre Bézier, a Bézier curve is a parametric function commonly used in computer graphics and modeling to create smooth shapes, paths, or to model animations. It is defined by a set of control points $\mathbf{P}$ and constructed using Bernstein basis polynomials. The general formulation of a Bézier curve is as follows:

$$\begin{aligned}\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} \mathbf{P}_i \quad 0 \leq t \leq 1 \\ &= \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \mathbf{P}_i \\ &= \sum_{i=0}^{n} b_{i,n}(t) \mathbf{P}_i\end{aligned} \tag{2.26}$$

where n is the order of the curve, $b_{i,n}$ is the Bernstein basis polynomial and $\mathbf{P}_i$ is the i^{th} control point. For a Bézier Curve of order n the number of control points $\#\mathbf{P}$ is equal to:

$$\#\mathbf{P} = n + 1 \tag{2.27}$$

One of the most valuable properties of the Bézier curve is its differentiability, with its derivative also being a Bézier curve of order $n-1$. To calculate the derivative, we begin by noting that since the control points $\mathbf{P}_i$ are constant and independent of time t, the calculation simplifies to differentiating the Bernstein basis polynomials:

$$\frac{\mathrm{d}}{\mathrm{d}t} b_{i,n}(t) = \dot{b}_{i,n}(t) = n\left(b_{i-1,n-1}(t) - b_{i,n-1}(t)\right) \tag{2.28}$$

We can then compute the derivative of the curve obtaining:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{B}(t) = \dot{\mathbf{B}}(t) = \sum_{i=0}^{n-1} b_{i,n-1}(t)\left[n\left(\mathbf{P}_{i+1} - \mathbf{P}_i\right)\right] \quad 0 \le t \le 1 \tag{2.29}$$

By defining

$$\mathbf{P'}_i = n\left(\mathbf{P}_{i+1} - \mathbf{P}_i\right) \tag{2.30}$$

we can express the previous formulation in a more compact form as:

$$\dot{\mathbf{B}}(t) = \sum_{i=0}^{n-1} b_{i,n-1}(t)\mathbf{P'}_i \quad 0 \le t \le 1 \tag{2.31}$$

It follows that, once the control points of a Bézier curve of order n are known, the control points of its derivative curve can be directly obtained. This greatly accelerates the computation process and reduces computational complexity.

Before proceeding with the analysis of how Bézier curves can be applied to parameterize the thrust trajectory of a jump, we must address the time interval over which the curve is defined. As seen in both Eq. 2.26 and Eq. 2.31, the curve is originally defined over a normalized time interval $0 \le t \le 1$. If we wish to consider a more general start time t_0 and end time t_f, we can express Eq. 2.26 in a more generalized form:

$$\begin{aligned}
\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i} \left(\frac{t-t_0}{t_f-t_0}\right)^i \left(1 - \frac{t-t_0}{t_f-t_0}\right)^{n-i} \mathbf{P}_i \quad t_0 \le t \le t_f \\
&= \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} \left(\frac{t-t_0}{t_f-t_0}\right)^i \left(1 - \frac{t-t_0}{t_f-t_0}\right)^{n-i} \mathbf{P}_i \\
&= \sum_{i=0}^{n} b_{i,n}\left(\frac{t-t_0}{t_f-t_0}\right)\mathbf{P}_i
\end{aligned} \tag{2.32}$$

In the case of the thrust phase, we define the time interval as follows:

$$t_0 = 0 \quad \text{and} \quad t_f = T_{th} \tag{2.33}$$

where T_{th} represents the total duration of the thrust phase. Substituting this into our parameterization, we can adjust the Bézier curve equation to accommodate this time interval, allowing the trajectory to be appropriately defined over the interval $[0, T_{th}]$ as:

$$\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}\left(\frac{t}{T_{th}}\right)\mathbf{P}_i \quad 0 \le t \le T_{th} \tag{2.34}$$

Since we have introduced a time scaling, the derivative must also be adjusted accordingly:

$$\begin{aligned}\mathbf{P'}_i &= \frac{n}{T_{th}}\left(\mathbf{P}_{i+1} - \mathbf{P}_i\right) \\ \dot{\mathbf{B}}(t) &= \sum_{i=0}^{n-1} b_{i,n-1}\left(\frac{t}{T_{th}}\right)\mathbf{P'}_i \quad 0 \le t \le T_{th}\end{aligned} \tag{2.35}$$

When selecting the order of the Bézier curve for our problem, we must consider the need to model both position and velocity. For any Bézier curve, the first and last control points correspond to the initial and final values of the curve. A third-order Bézier curve offers an ideal balance for modeling the position trajectory, as it provides four control points. The two intermediate control points can be used to shape the velocity profile, since all the four same control points are later used to compute the derivative, and thereby the velocity trajectory.

Before delving into the application of the Bézier curve for thrust phase parameterization, it is helpful to present the explicit forms of both the cubic Bézier curve and its derivative. For clarity in notation, we will define the normalized time $\hat{t}$ as follows:

$$\hat{t} = \frac{t}{T_{th}} \quad 0 \le t \le T_{th} \tag{2.36}$$

The explicit forms for both the cubic Bézier curve and its corresponding quadratic derivative are as follows:

$$\mathbf{B}(t) = (1-\hat{t})^3\mathbf{P}_0 + 3(1-\hat{t})^2\hat{t}\mathbf{P}_1 + 3(1-\hat{t})\hat{t}^2\mathbf{P}_2 + \hat{t}^3\mathbf{P}_2 \tag{2.37}$$

$$\begin{aligned}\dot{\mathbf{B}}(t) &= \frac{3}{T_{th}}(1-\hat{t})^2(\mathbf{P}_1 - \mathbf{P}_0) + \frac{6}{T_{th}}(1-\hat{t})\hat{t}(\mathbf{P}_2 - \mathbf{P}_1) + \frac{3}{T_{th}}\hat{t}^2(\mathbf{P}_3 - \mathbf{P}_2) \\ &= (1-\hat{t})^2\mathbf{P'}_0 + 2(1-\hat{t})\hat{t}\mathbf{P'}_1 + \hat{t}^2\mathbf{P'}_2\end{aligned} \tag{2.38}$$

2.7.2 Thrust Trajectory Parametrization

Now that we have introduced and defined the parametric Bézier curve, we need to understand how this mathematical object can be employed to parameterize the thrust trajectory. From the previous section, we have established that the lift-off configuration is a crucial factor in determining the jump's outcome. While this configuration will be the primary action governed by our RL policy, it is now necessary to model the trajectory that connects the starting position to the lift-off configuration determined by our agent. This is where the Bézier curve proves to be particularly useful.
As mentioned earlier, the initial and final control points of the Bézier curve correspond to the initial and final values that the curve will assume. We have provided the explicit forms of both the third-order Bézier curve and its derivative. The next step is to use these definitions from Eq. 2.37 and Eq. 2.38, and, by applying the boundary conditions for the initial and lift-off CoM positions $\mathbf{c}_0$,$\mathbf{c}_{lo}$ as well as the initial and lift-off CoM velocities $\dot{\mathbf{c}}_0$,$\dot{\mathbf{c}}_{lo}$, compute the control points $\mathbf{P}_i$.

$$\begin{cases}\mathbf{P}_0 = \mathbf{c}_0 \\ \mathbf{P}_3 = \mathbf{c}_{lo} \\ \mathbf{P'}_0 = \frac{3}{T_{th}}(\mathbf{P}_1 - \mathbf{P}_0) = \dot{\mathbf{c}}_o \\ \mathbf{P'}_1 = \frac{3}{T_{th}}(\mathbf{P}_2 - \mathbf{P}_1) \\ \mathbf{P'}_2 = \frac{3}{T_{th}}(\mathbf{P}_3 - \mathbf{P}_2) = \dot{\mathbf{c}}_{lo}\end{cases} \tag{2.39}$$

From system 2.39, we observe that, as previously mentioned, only the control points $\mathbf{P}_1$ and $\mathbf{P}_2$ actively contribute to defining the shape of the position trajectory through their imposed relationship with the velocity trajectory. In fact, these are the only two remaining parameters that need to be determined by solving the system.
We begin by calculating the value of $\mathbf{P}_1$:

$$\begin{aligned} \mathbf{P}_0 = \mathbf{c}_0 \quad \mathbf{P'}_0 =& \frac{3}{T_{th}}(\mathbf{P}_1 - \mathbf{P}_0) = \dot{\mathbf{c}}_o \\ \mathbf{P}_1 =& \frac{T_{th}}{3}\dot{\mathbf{c}}_0 + \mathbf{P}_0 \\ =& \frac{T_{th}}{3}\dot{\mathbf{c}}_0 + \mathbf{c}_o \end{aligned} \tag{2.40}$$

Next, we proceed to calculate $\mathbf{P}_2$:

$$\begin{aligned} \mathbf{P}_3 = \mathbf{c}_{lo} \quad \mathbf{P'}_2 =& \frac{3}{T_{th}}(\mathbf{P}_3 - \mathbf{P}_2) = \dot{\mathbf{c}}_{lo} \\ \mathbf{P}_2 =& -\frac{T_{th}}{3}\dot{\mathbf{c}}_{lo} + \mathbf{P}_3 \\ =& -\frac{T_{th}}{3}\dot{\mathbf{c}}_{lo} + \mathbf{c}_{lo} \end{aligned} \tag{2.41}$$

Now that we have determined the two missing control points, we can specify all the control points for both the position and velocity trajectories of the thrust phase:

Position	Velocity
$\mathbf{P}_0 = \mathbf{c}_0$	$\mathbf{P'}_0 = \dot{\mathbf{c}}_0$
$\mathbf{P}_1 = \frac{T_{th}}{3}\dot{\mathbf{c}}_0 + \mathbf{c}_0$	$\mathbf{P'}_1 = \frac{3}{T_{th}}(\mathbf{P}_2 - \mathbf{P}_1)$
$\mathbf{P}_2 = -\frac{T_{th}}{3}\dot{\mathbf{c}}_{lo} + \mathbf{c}_{lo}$	$\mathbf{P'}_2 = \dot{\mathbf{c}}_{lo}$
$\mathbf{P}_3 = \mathbf{c}_{lo}$	

Table 2.1: Control point values for the two Bézier curves used in thrust trajectory parametrization

An observation can be made for the specific case where the robot's initial velocity is zero. In this scenario, the first term of $\mathbf{P}_1$ cancels out, making this control point identical to $\mathbf{P}_0$. Consequently, only $\mathbf{P}_2$ actively influences and stretches the curve profile (see Fig. 2.7).
In this work, for the sake of simplicity, the initial configuration will always assume zero velocity.
Despite this simplification, the formulation remains general, and future work will include scenarios where the robot has an initial velocity different from zero, incorporating such conditions into the training phase.

Having described how the thrust trajectory is parameterized using engineering knowledge, we can now proceed to analyze how the lift-off configuration is obtained through GRL. This phase's parameterization is therefore essential and serves as a key concept in the GRL approach, as it reduces complexity and eliminates the need for complex reward shaping to model the desired behavior of the robot during this phase.

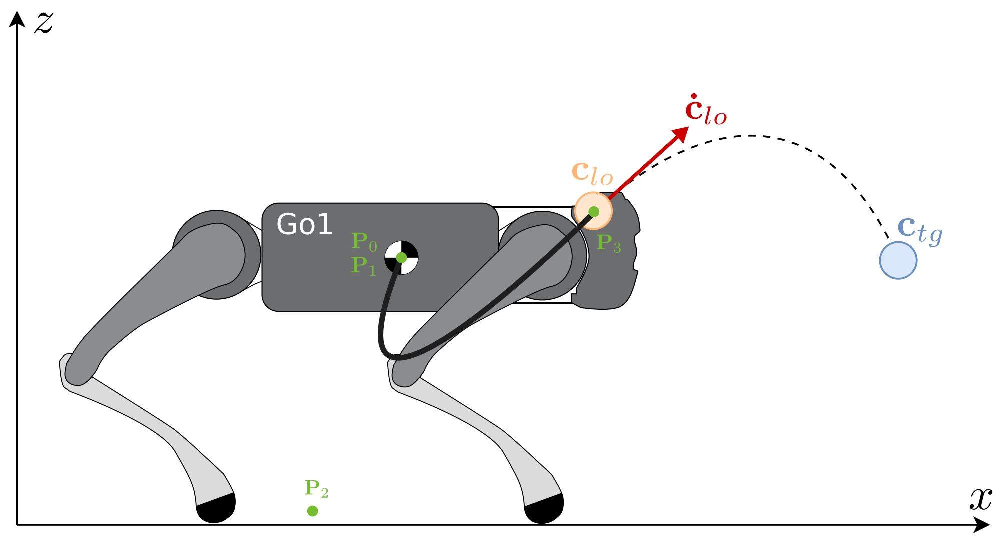


Figure 2.7: Thrust trajectory computed using a Bézier curve (green dots represent the control points

3 Approach

3.1 Efficient Reinforcement Learning for 3D Jumping Monopods

Throughout the state-of-the-art overview and the background chapter, we have frequently referenced our previous work [39], which proposed the use of GRL to efficiently learn a strategy for generating 3D jumps on a monopod. This preliminary work serves as a foundational basis for the approach presented in this thesis. Before addressing the advancements and extensions introduced in this research for quadrupeds, we will first provide an overview of the ideas, strategies, and results developed in the prior work, as they are crucial for understanding the subsequent developments.
The old project was initiated as a proof of concept to demonstrate that RL can be used to perform real-time jumps on legged robots while also incorporating engineering and domain-specific knowledge to enhance training efficiency without requiring millions of episodes (that's what GRL is about). To demonstrate the superiority of the proposed method, a comparison was made between the GRL-based approach, the E2E-based approach, and a classical TO-based method, evaluating both the speed and accuracy of the generated trajectories.

Quadruped robots are challenging systems to control due to their 12 actuated DoFs. For simplicity, this work considers a simplified yet realistic scenario: a monopod robot whose base link is supported by a set of three passive prismatic joints. These joints form a virtual kinematic chain that controls the position of the base link relative to the world frame W, while preventing any changes in its orientation. The study, therefore, focuses solely on the linear aspects of the jumping problem, neglecting the angular dynamics. The robotic platform used is a single leg of the open-source quadruped Solo [58], as illustrated in Fig. 3.1. Henceforth, we will refer to that as a monopod.

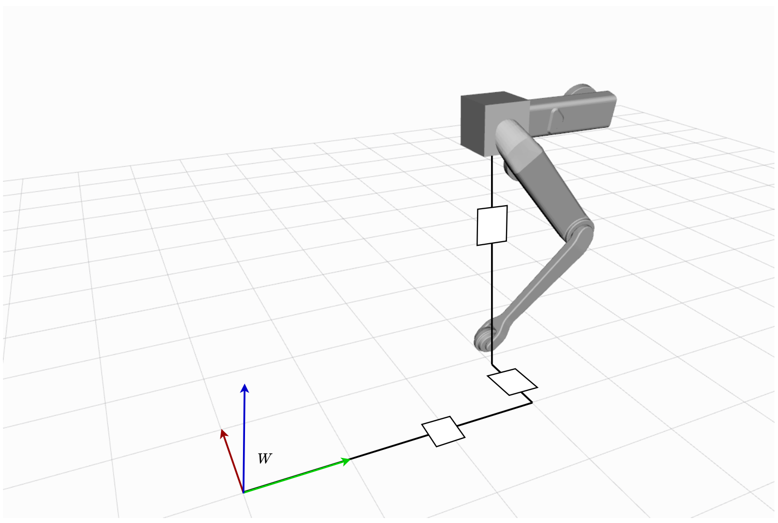


Figure 3.1: Rviz visualization of the monopod used in the previous work, including the passive prismatic joints that connect the world frame W to the base link

3.1.1 The Action space

In the Ballistic Problem section 2.6, while focusing on the linear aspect of the problem, we highlighted that once the robot is airborne, no additional thrust can be produced. As a result, the entire trajectory follows ballistic motion due to the conservation of momentum. We also pointed out that the entire jump is determined solely by the CoM lift-off configuration, which is defined by the position and linear velocity $(\mathbf{c}_{lo}, \dot{\mathbf{c}}_{lo})$. Since the flight trajectory is ballistic, this means that the entire jump occurs within the vertical plane that connects the CoM's starting position $\mathbf{c}$ to its target position $\mathbf{c}_{tg}$.
Furthermore, we discussed how finding the optimal lift-off configuration is challenging due to the high-dimensional optimization problem, which is constrained by a large number of factors. This is why the lift-off configuration can be considered as part of the RL action space.

In the Trajectory Parameterization section 2.7, we showed that given a lift-off configuration and an initial configuration defined by the initial position and initial linear velocity, the position and velocity trajectories can be represented by a third-order and second-order Bézier curve, respectively, using the control points outlined in Tab. 2.1. The only remaining free parameter in defining the curve is the thrust time T_{th}, which plays a key role in shaping the thrust trajectory. Specifically, as shown in Fig. 3.2, for a fixed initial and lift-off configuration, a longer thrust time T_{th} results in a longer overall curve, while a shorter thrust time produces a more compact trajectory. Therefore, we can include the thrust time T_{th} as part of the action space in our RL policy.

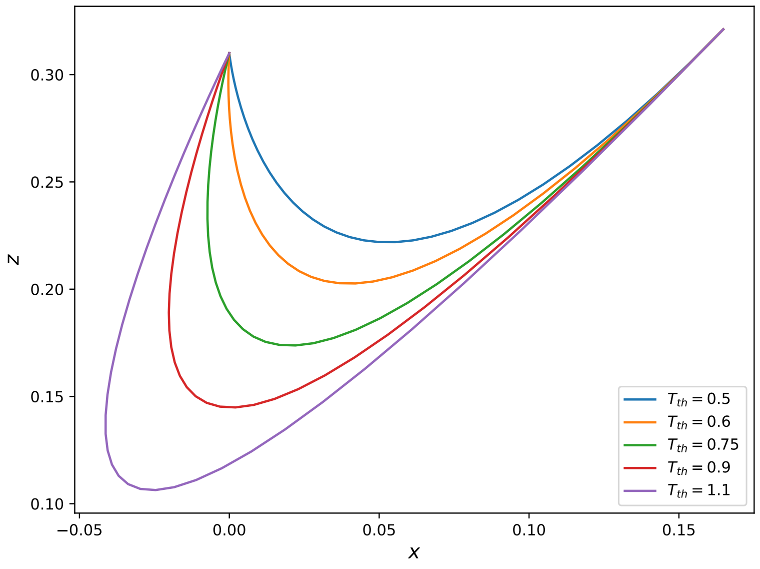


Figure 3.2: Rviz visualization of the monopod used in the previous work, including the passive prismatic joints that connect the world frame W to the base link

In conclusion, the action space is defined as:

$$\begin{aligned} \mathbf{a} &= (T_{th}, \mathbf{c}_{lo}, \dot{\mathbf{c}}_{\mathbf{lo}}) \in \mathbb{R}^7 \\ &= (T_{th}, \mathbf{c}_{lo,x}, \mathbf{c}_{lo,y}, \mathbf{c}_{lo,z}, \dot{\mathbf{c}}_{lo,x}, \dot{\mathbf{c}}_{lo,y}, \dot{\mathbf{c}}_{lo,z}) \end{aligned} \tag{3.1}$$

The dimensionality of the action space significantly impacts performance by influencing the size of the exploration space. A smaller action space reduces the exploration area, simplifying the complexity of the mapping and speeding up the learning process. One approach to reducing the action space is to express both the CoM lift-off position $\mathbf{c}_{lo}$ and linear velocity $\dot{\mathbf{c}}_{lo}$ in spherical coordinates.
Due to the ballistic nature of the flight phase, and since the entire jump trajectory lies within the plane connecting the initial and target locations, we can leverage the fact that the yaw angle φ (which defines the orientation of the jumping plane in the $X - Y$ frame) remains constant throughout the jump. This angle can be computed analytically from the initial and target positions to further simplify

the parametrization.

Given this, the coordinates of the lift-off configuration can be restricted to a convex, two-dimensional space:

$$\begin{cases} \mathbf{c}_{lo,x} = r\cos(\theta)\cos(\varphi) \\ \mathbf{c}_{lo,y} = r\cos(\theta)\sin(\varphi) \\ \mathbf{c}_{lo,z} = r\sin(\theta) \end{cases} \quad \begin{cases} \dot{\mathbf{c}}_{lo,x} = r_v\cos(\theta_v)\cos(\varphi) \\ \dot{\mathbf{c}}_{lo,y} = r_v\cos(\theta_v)\sin(\varphi) \\ \dot{\mathbf{c}}_{lo,z} = r_v\sin(\theta_v) \end{cases} \tag{3.2}$$

As shown in Fig. 3.3, the lift-off position vector $\mathbf{c}_{lo}$ is characterized by the extension radius r, the pitch angle θ, and the pre-computed yaw angle φ. Similarly, the lift-off velocity vector $\dot{\mathbf{c}}_{lo}$ is described by its magnitude r_v, the pitch angle θ_v, and the same yaw angle φ as the position vector.

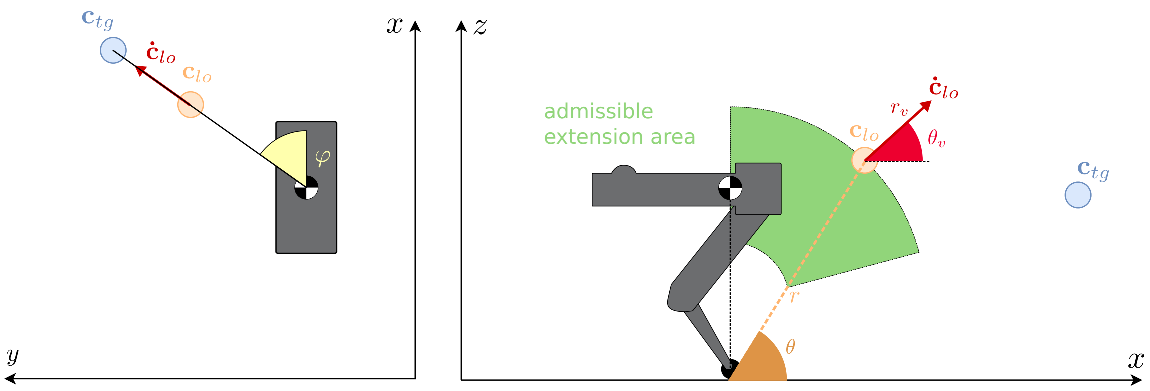


Figure 3.3: Top and side views of the action in spherical coordinates

Thanks to this coordinate representation change and the imposition of the fixed yaw angle φ, the action dimension is now reduced to 5, and is defined as:

$$\mathbf{a}_{\text{old}} = (T_{th}, r, \theta, r_v, \theta_v,) \in \mathbb{R}^5 \tag{3.3}$$

To narrow the search space for these parameters, we can constrain their range of values by applying domain knowledge, ensuring that the actions always produce physically feasible trajectories without compromising optimality.
The ranges and the method by which they are selected are as follows:

- **Time of thrust** (T_{th}): The entire thrust phase must be executed quickly, depending on the desired jump distance. Therefore, this parameter has a range of: $[0.1, 1]$ seconds.
- **Extension radius** (r): The range of this value must prevent boundary singularities due to over-extension and avoid complete leg retraction. Its range depends on the robot's characteristics, so we define it generally as: $[r_{\min}, r_{\max}]$ meters.
- **Position pitch angle** (θ): This value helps avoid excessive foot slippage and unnecessary force exertion during the thrust phase by controlling the vertical and horizontal components of the jump. The range is set to allow configurations from fully vertical to moderately inclined, that would in any case cause excessive slippage or falls: $\left[\frac{\pi}{4}, \frac{\pi}{2}\right]$ radians. However, we will explicitly enforce friction constraints with a specific reward
- **Velocity magnitude** (r_v): This parameter determines how far the robot will jump and must be constrained by an upper limit. Since the exact value for which the action is always feasible is unknown, this bound serves as an ideal limit to guide the policy and restrict exploration: $[0.1, 4]$ meters per second.
- **Velocity pitch angle** (θ_v): Similar to the position pitch angle, this value guides the jump toward the target. It must be positive and bounded between a fully vertical configuration and

an inclined one that avoids excessive horizontal velocity: $\left[\frac{\pi}{6}, \frac{\pi}{2}\right]$ radians.

This action formulation simplifies the description of the entire jump task into just a few parameters, effectively eliminating issues related to task sparsity since the episode is executed with a single action. In contrast to E2E approaches, our GRL method leverages domain-specific knowledge to construct a problem that does not require millions of training episodes to learn an optimal trajectory from scratch. This also has a significant impact on reward function design, as it simplifies the reward structure. As we will discuss shortly, the reward function does not contain complex terms and closely resembles the formulation of an optimal control problem.

3.1.2 A Physically Informative Reward Function

The reward function is the sole indicator that evaluates how effective an action is in achieving the task objective, making its appropriate design crucial for success. Similar to the action formulation, the reward function can be used as a means to inject prior knowledge into the learning process. In this work, the reward function is engineered much like an optimal control problem, where it penalizes violations of the system's physical constraints while maximizing a target reward for landing near the designated target location.

The constraints that must be adhered to throughout the entire thrust phase are referred to as path constraints. Any violation of these constraints can be penalized using a linear activation function, defined as:

$$A(x, \underline{x}, \overline{x}) = \left|\min(x - \underline{x}, 0) + \max(x - \overline{x}, 0)\right| \tag{3.4}$$

where $\underline{x}$ and $\overline{x}$ represent the lower and upper bounds, respectively.
As shown in Fig. 3.4, the value of this activation function is zero when the input remains within the allowed range and reflects the magnitude of the violation otherwise.

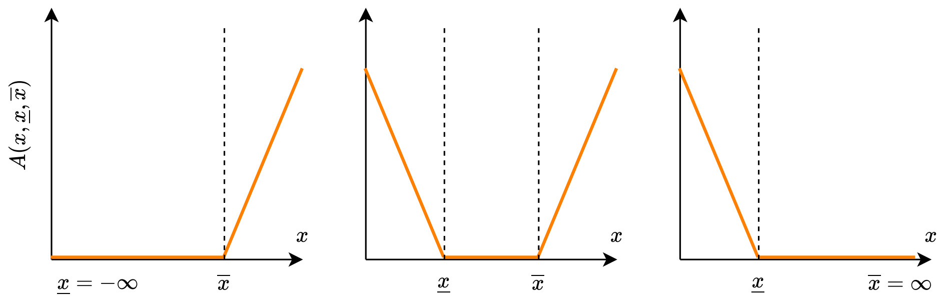


Figure 3.4: Activation function output in three possible scenarios: no lower bound, both lower and upper bounds, and no upper bound

The path constraints are checked at each time step, and any violation costs are evaluated and accumulated into what is referred to as the feasibility cost C_f.
The path constraints are as follows:

- **Joint position and velocity range**: The joint kinematic limits must not be exceeded:

$$\underline{q}_i \leq q_i(t) \leq \overline{q}_i, \quad \underline{\dot{q}}_i \leq \dot{q}_i(t) \leq \overline{\dot{q}}_i$$

  where q_i refers to the i-th joint of the leg

- **Torque range**: Similarly, torque limits must be respected:

$$\underline{\tau}_i \leq \tau_i(t) \leq \overline{\tau}_i$$

- **Friction cone**: To avoid slippage, the tangential component of the contact force $\|\mathbf{F}_{x,y}\|$ must remain within the friction cone defined by the vertical contact force component $\mathbf{F}_z$ and the terrain's static friction coefficient μ:

$$\|\mathbf{F}_{x,y}\| \leq \mu \mathbf{F}_z$$

- **Unilaterality**: In legged robots, legs are only allowed to push on the ground, not pull. This corresponds to having the vertical component (for flat terrains) of the contact force always positive:

$$\mathbf{F}_z \geq 0$$

- **Singularity**: During the thrust phase, where contact is assumed to be maintained, the robot must remain within its admissible extension area. Exceeding this area may result in reduced mobility and high joint velocities in the IK computation. Although this risk is minimized by the defined action range, it is not entirely eliminated. In this work, if a singularity configuration occurs, the episode is stopped and a high penalty is applied.

In addition to the path constraint costs, additional costs are introduced to promote specific behaviors in the policy:

- **Lift-off tracking error** (C_{lo}): While both the extension radius and velocity magnitude are selected within feasible ranges for the robot, this does not guarantee that the desired thrust trajectory will be fully achieved. To address this, the lift-off tracking error is introduced, ensuring that the trajectory performed by the robot is accurately tracked by the low-level controller by minimizing any deviations from the desired lift-off configuration.

- **No Touchdown penalty** (C_{td}): This penalty is applied to prevent the monopod from remaining stationary during short or in-place jumps. It is triggered when an episode does not result in a touchdown, defined as the moment when the feet reestablish contact with the ground after being airborne.

- **Physical feasibility check** (C_{ph}): This is a key feature of the task formulation and serves as a safety measure to determine whether a jump is physically feasible based on the proposed trajectory parameters and target location. As we will discuss later, in the absence of parallelization, to also speed up the training, this check is used to abort an episode at the start if the vertical velocity is insufficient to reach the target height, resulting in a high penalty.
  The feasibility check is conducted as follows:

$$\begin{aligned} T_{fup} &= \frac{\dot{\mathbf{c}}_{lo,z}}{g} \\ \overline{\mathbf{c}}_z(T_{fup}) &= \mathbf{c}_{lo,z} + \dot{\mathbf{c}}_{lo,z} T_{fup} + \frac{1}{2}(-g) T_{fup}^2 \\ &= \mathbf{c}_{lo,z} + \frac{1}{2}\frac{\dot{\mathbf{c}}_{lo,z}^2}{g} \end{aligned}$$

  where T_{fup} is the flight time required to reach the apex, and $\overline{\mathbf{c}}_z(T_{fup})$ is the predicted apex height. The jump is considered unfeasible if $\mathbf{c}_{tg,z} > \overline{\mathbf{c}}_z(T_{fup})$.

Finally, the key component of the reward function is the landing target reward function (R_{lt}), which measures how close the landing location is to the desired target location. This reward is calculated at the CoM level for consistency with the thrust trajectory; however, an equivalent formulation could be applied at the foot level, as the CoM is expected to align with the position of the foot at the end of the jump.

The landing target reward function is defined as:

$$R_{lt}(\mathbf{c}, \mathbf{c}_{tg}) = \frac{\beta}{\alpha \|\mathbf{c} - \mathbf{c}_{tg}\| + \epsilon} \tag{3.5}$$

where α is a hyperparameter that encourages jumps with minimal error, β is a hyperparameter that bounds the maximum value of the function, and ϵ is a small value added to ensure that no division by zero occurs.

With all components defined, the total reward function is formulated as the landing target reward minus the sum of all constraint costs:

$$R = 1_{\mathbb{R}+} \left[R_{lt}(\mathbf{c}, \mathbf{c}_{tg}) - \sum C_i \right] \tag{3.6}$$

This reward function is clamped to $\mathbb{R}^+$ using the indicator function $1_{\mathbb{R}+}$, ensuring that the reward remains non-negative. This reward shaping encourages actions where the penalties do not significantly affect the jump outcome, thereby promoting constraint satisfaction. The goal of the policy, in fact, is to maximize the reward, which inherently drives the minimization of all penalty costs associated with the task.

3.1.3 Implementation Details

When the development of this work began, the powerful Isaac-Lab framework had not yet been introduced, and only an early version of the Isaac-Gym simulator was available. As mentioned in the Simulative Technology section 2.4, several alternative solutions existed, but we chose to base our work on the Locosim framework [56]. This decision was motivated by the fact that Locosim offers a robust foundation for controlling both fixed-base and floating-base robots. It has been thoroughly tested and, being fully open-source, presents an excellent solution for our needs. Additionally, integrating RL into Locosim not only benefits this project but also lays a foundation for future works and academic projects.

Locosim is entirely based on a ROS control architecture, with a C++ roscontrol node that interfaces with a Python ROS node to enable interaction with the Gazebo simulator. However, one major drawback of Locosim's design is the lack of easy control parallelization, limiting RL from fully leveraging parallel training. Another key limitation is the computational inefficiency of the Gazebo simulator, which, in our case, struggled to achieve real-time rates higher than 0.3. As a result, simulating an episode took a considerable amount of time, making each episode critically important for training.

To fully leverage the collected data without discarding it after each training step, we employed the state-of-the-art Twin Delayed DDPG (TD3) algorithm [59], which was used in the previous monopod work. The Background chapter does not cover the details of TD3 since the new quadruped work uses PPO instead. As an off-policy algorithm, TD3 stores all collected data in a replay buffer, allowing state-action trajectories to be sampled multiple times to refine and optimize the policy over time.

To evaluate and compare the performance of our GRL approach, a classical TO approach and an E2E approach were developed similarly to what has been described so far. The E2E approach, instead of generating the parameters for thrust trajectory parameterization, directly produces joint displacements from a default configuration at each time step. These are directly set as a reference for the low level controller. Several adjustments were made to the reward function to improve stability, such as using lower frequency actions and reducing abrupt changes in joint trajectories between consecutive time steps. The state space was also modified to include the current joint positions at the time of observation. To align with the state-of-the-art in RL algorithms for robot control, PPO was used for this case.

For the non-linear TO approach instead, an optimal control strategy based on Feasible Differential Dynamic Programming (FDDP) [60] was employed. This strategy provides an efficient solution for whole-body control (WBC) by exploiting the sparse structure of optimal control. In this case, the trajectory was discretized into N successive knots with a timestep of both $dT = 1 or 2\,\text{ms}$, and the maximum number of iterations was set to 500. Joint torques were used as decision variables in this case.

3.1.4 Simulation Results

Having provided an overview of the work, we will now present the results obtained from this approach. This is crucial to demonstrate why GRL provides an effective and straightforward solution to the jumping problem for RT planning applications, and to further support the motivation behind this thesis, which focuses on extending the previous proof-of-concept work.
The training region, where the target jump locations are sampled, is a cylindrical area with a radius of 0.65m, extending in height from 0.25m (the nominal height of the monopod in the standing configuration) up to 0.5m. This setup is designed to push the robot to its system limits. The test region is expanded by 20% to evaluate the generalization of the learned policy. The test area is discretized into six different height levels, with a total of 726 uniformly spaced target positions.

To quantify the quality of a jump, we introduce the Relative Percentual Error (RPE) metric, which is defined as the distance error between the touchdown point $\mathbf{c}_{td}$ and the target location $\mathbf{c}_{tg}$, divided by the desired jump distance from the initial position $\mathbf{c}_0$:

$$RPE = \frac{\|\mathbf{c}_{tg} - \mathbf{c}_{td}\|}{\|\mathbf{c}_{tg} - \mathbf{c}_0\|} \tag{3.7}$$

The feasibility region is defined as the area where the agent can perform an accurate landing, i.e., an $RPE \leq 10\%$. As an additional evaluation criterion, the average computation time for generating the thrust trajectory is also considered.

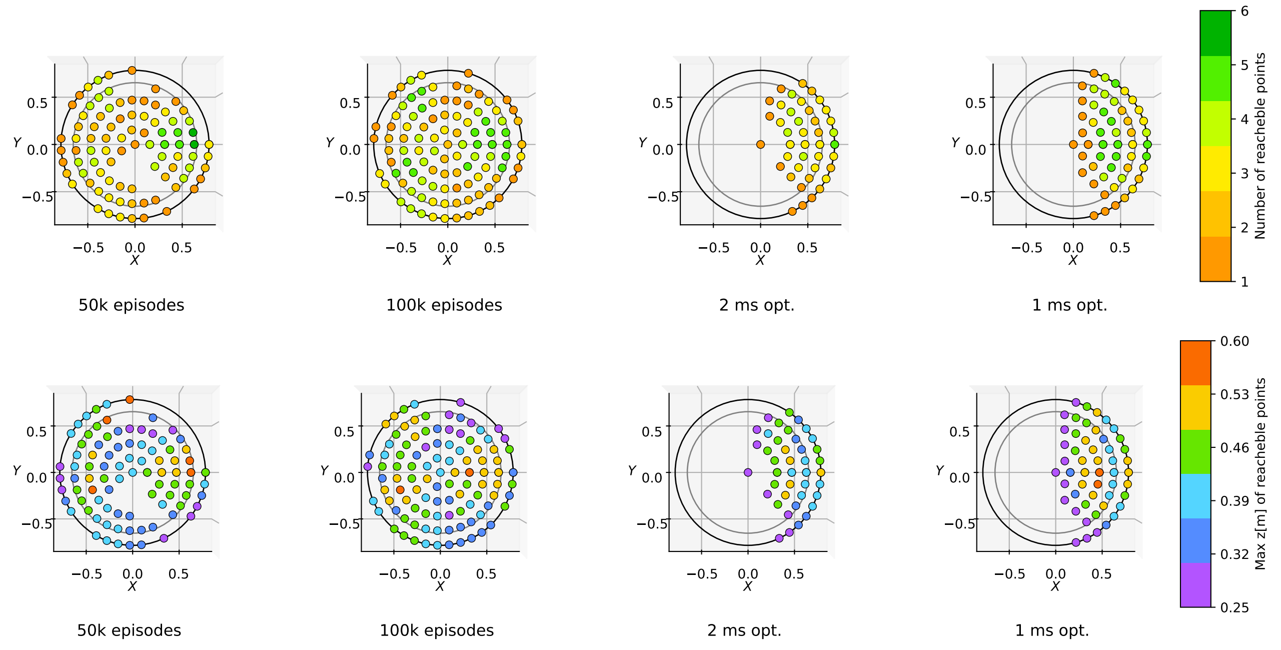


Figure 3.5: Top-view of the feasibility region: (first 4 plots in the row) showing different numbers of episodes during the training phase, and (last 2 plots in the row) representing the baseline FDDP case

In Fig. 3.5, the feasibility region is shown for both the GRL and FDDP approaches. The first row of plots represents the maximum number of reachable points (discretized into six height levels), while the second row shows the maximum reachable height for each position within the feasible set. The E2E approach is not included due to its extremely poor performance (with only a few points being reached out of the entire set).

While E2E approaches can achieve impressive results in locomotion tasks such as walking, they tend to struggle significantly with jumping tasks and require substantial engineering effort and staged learning strategies [45]. It is immediately evident that while the GRL approach achieves reasonable accuracy across the entire range of directions and distances, the FDDP approach performs poorly, particularly for targets located behind the robot.
The discrepancy in performance between forward and backward jumps using the FDDP method can be attributed to its limited generalization capabilities, due to the asymmetry of the leg, which often results in being trapped in poor local minima. This issue could potentially be mitigated by splitting the optimization problem for forward and backward regions, carefully tuning the weights, or augmenting the decision variables to include jump phase timings—though this would significantly increase computational costs.

In Tab. 3.1, the mean RPE values and the computation time required to plan the trajectory for both the forward and backward regions are reported for each approach. From these results, it is clear that the GRL approach outperforms the classical TO approach in both accuracy and, notably, computation time, where the GRL method is several orders of magnitude faster. This initial work strongly motivates our decision to extend the approach to a more complex quadruped robot, building upon the ideas and methodology presented here.

	mean RPE - FDDP	mean RPE - GRL	time - FDDP	time - GRL
Front region	16.5%	16%	2.13 - 4.3 s	0.7 ms
Back region	52%	18%	2.35 - 4.3 s	0.7 ms

Table 3.1: Summary of the results obtained in terms of jump accuracy and computation time required to plan the jump action

3.2 From Monopod to Quadruped

Up until now, the focus has been on the linear aspects of the problem, from thrust trajectory parameterization to policy design and its simplification for a monopod robot. However, extending these approaches to include the angular domain is crucial to fully harness the capabilities of quadrupeds, and more generally, legged robots.

As discussed in the state-of-the-art section 1.2, a common simplification used in TO approaches to reduce computational complexity is the use of a reduced 5-DoFs representation of the quadruped in the sagittal plane. However, this is only suitable for planar jumps and not for 3d jumps. Another common approach involves directly optimizing the GRFs and incorporating also the robot's orientation in the dynamics, but this often introduces artificial constraints, such as fixed contact timing. This assumption is taken to avoid the dynamics of the system being hybrid when it comes to planning across the mode transition related to the landing-take-off phase.
In this research, we consider the complete robot model and, as a key advancement, extend the whole method to include the angular components of the jump. This unlocks the full potential of the system and proposes a novel and efficient approach to solving the jumping problem in quadrupeds.

As discussed in the jump taxonomy section 2.5, controlling the robot's orientation during the thrust phase is crucial for distributing contact forces, preventing imbalances, and avoiding slippage. By managing the orientation, we can also control the angular velocity and, consequently, the angular momentum at the lift-off, which directly influences the airborne phase, as this phase is governed purely by momentum conservation. This capability enables dynamic maneuvers such as in-place twists for rapid reorientation or to regulate the body's angular velocity to optimize landing (e.g. reducing impacts or achieving better landing configuration). For instance, a pitched-up landing can improve impact absorption by initially dispersing force on the rear legs, followed by cushioning the remaining impact with the front legs.

3.2.1 Managing the Landing

In the monopod work [39], episodes terminated upon touchdown detection, as controlling leg balance was outside the scope. However, in this work, to develop a policy that can be directly applied in the real world, the episode continues until a maximum timeout is reached. This means that, after reestablishing contact with the ground, the robot must balance itself and return to a nominal configuration, compensating for the compression caused by the trunk inertia.
Landing control is a complex task, and specific strategies exist to address it [5], but for the sake of simplicity, we have streamlined this process in our approach. During the thrust phase, stiffness is tuned into the PD controller in order to achieve a reasonable tracking accuracy together with WBC gravity compensation. Once the thrust phase ends, the stiffness is reduced by a division factor, the WBC is deactivated, and the legs are retracted. Upon reaching the apex (the maximum height of the jump, when the z-component of the linear velocity changes sign and becomes negative) the legs are extended back to a nominal default configuration. When touchdown is detected via a threshold applied to the measured GRFs, the WBC is reactivated to again compensate for gravity.

To avoid discontinuities in torque during the configuration change in the airborne phase, a Cubic Hermite Spline Interpolation is used to smooth the joint reference when changing configuration from extended to retracted and viceversa:

$$\begin{gathered} q(t) = h_{00}(t)q_s + h_{10}(t)m_0 + h_{01}(t)q^d + h_{11}(t)m_1 \\ h_{00}(t) = 2t^3 - 3t^2 + 1 \quad h_{10}(t) = t^3 - 2t^2 + t \\ h_{01}(t) = -2t^3 + 3t^2 \quad h_{11}(t) = t^3 - t^2 \end{gathered} \tag{3.8}$$

where q_s is the starting joint configuration, q^d is the desired joint configuration, and m_0 and m_1 are parameters indicating the initial and final slopes. The time t is normalized to the range $0 \leq t \leq 1$, with the entire interpolation duration set to 0.1 seconds, similar to the normalization used in the Bézier trajectory.

Previously, a detailed study on the landing control was conducted by Roscia et al. [5] using the same Go1 robotic platform. Their method proposed a real-time reactive Landing Controller (LC) capable of handling substantial horizontal velocities while ensuring safe landings, without requiring precise estimates of flight time or distance to the landing surface. The approach utilizes proprioceptive data and applies principles such as capturability and Zero Moment Point (ZMP) to stabilize the robot after touchdown, preventing bouncing, slippage, and trunk collisions. This work was implemented within the Locosim framework, which, as previously mentioned, was used for the sim-to-real control system. Drive by the motivation to keep our approach simple we decided to not include the LC as part of the simulation in the training phase, we will demonstrate in simulation the integration of this landing controller with our jump strategy, which focuses primarily on the thrust phase. We will compare the impact of using this landing controller against our simplified strategy.

In a future extension of this work, we plan to introduce a separate NN trained using an E2E approach, specifically tasked to adjust the joint configuration at each time step. This would help reorient the robot mid-air and significantly improve the robustness of the landing behavior mimic the intervention of a LC, building on the simplified version described above, while ensuring that the entire approach remains applicable in RT scenarios.

3.2.2 Jump Strategy for Quadrupeds

In the previous work, jumping techniques were not a critical consideration due to the absence of multiple legs. However, for quadrupeds, the choice of jumping technique is key in defining the control strategy. Different animals have evolved various jumping strategies depending on the purpose of the action, such as overcoming obstacles, attacking prey, or escaping predators. One of the primary jumping strategies is leaping, where the hind limbs generate the main thrust, enabling the animal to cover long distances or heights. In the leaping technique, the animal initially pushes the ground with all four legs. As it propels forward, it pitches its body up so that only the hind legs remain in contact with the ground, and the final push is executed by these remaining two legs. Despite its prevalence in the animal kingdom, this jumping mechanic adds complexity to the thrust phase definition and, consequently, to the control strategy.

As explained, the thrust phase of a leaping jump can be divided into two distinct periods: the first involves all four legs pushing, while the second involves only the two remaining legs. This introduces the need for a split in the thrust trajectory parameterization and requires two distinct control strategies for the robot. The first strategy, as described in the robot control section, is used when all legs are in contact with the ground. In contrast, the second strategy must account for reduced mobility and the added complexity that arises from the underactuation when two legs are lifted off the ground.

In this work, we have chosen to adopt a jumping technique known as pronking (or stotting), a behavior typically observed in springboks. Unlike leaping, in this strategy all four legs push simultaneously throughout the entire thrust phase, lifting the body with all legs at once, as illustrated in Fig. 3.6.[1] By using this technique, the thrust phase remains a single continuous phase, allowing for a straightforward parametrization.

Since all four feet are in contact with the ground throughout the thrust phase, the control strategy for tracking the desired Cartesian trajectory and compensating for gravity remains the same as defined in section 2.2.1. There is no need to modify the Jacobian matrix J_{cb} (eq. 2.4) by zeroing columns relative to the joints of the legs in the air, as contact is maintained at all times. This is enforced by ensuring the fulfillment of both the friction cone and unilaterality constraints during training (see section 3.1.2).

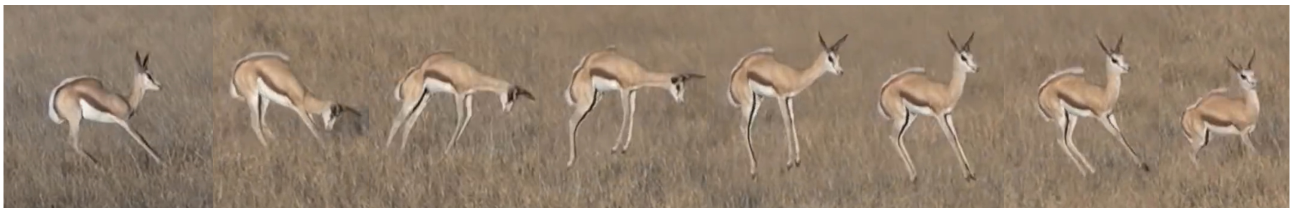


Figure 3.6: Image sequence of a springbok performing a pronking jump

3.2.3 Angular Extension of Thrust Trajectory Parametrization

In section 2.7, we discussed how the thrust trajectory can be parameterized using Bézier curves. As demonstrated in our previous work with the monopod, this approach has proven to be efficient and effective compared to classical E2E and TO methods. However, the results presented so far do not account for the angular aspect of the problem, as orientation has been neglected.

Recalling what was defined earlier in the ballistic problem section 2.6, the full CoM configuration of the robot is represented by the pair of vectors $(\mathbf{s}, \dot{\mathbf{s}})$, where $\mathbf{s}$ includes the CoM position $\mathbf{c}$ and orientation $\mathbf{\Phi}$, and $\dot{\mathbf{s}}$ includes the CoM linear velocity $\dot{\mathbf{c}}$ and angular velocity $\dot{\mathbf{\Phi}}$.

[1]Sequence of images generated from the video "pronking springbok": `https://www.youtube.com/watch?v=iIigVlcxwps`

The orientation is parametrized using Euler angles, while the angular velocity is given in terms of Euler rates.

Starting from the explicit forms of both the cubic Bézier curve (2.37) and its quadratic derivative (2.38), we expressed the fulfillment of the boundary condition as a linear system 2.39, which collects all boundary conditions. Once solved, this system provides the control points of the position trajectroy, as reported in Table 2.1.
Incorporating orientation into the thrust trajectory presents a secondary independent parameterization task. Using the tangent of the already computed Bézier curve to define orientation would in fact impose artificial constraints that limit the flexibility of the method, also, excessive pitch components would result easily in unfeasible solutions. Having a separate trajectory for the orientation, as previously mentioned, enables complex and dynamic maneuvers, such as in-place twist jumps for rapid orientation changes. Additionally, it allows for fine-tuned adjustments to the robot's airborne orientation, optimizing during the thrust phase its position in preparation for landing impact.
Therefore, following the same reasoning behind using Bézier curves for the position trajectory, we propose using a separate Bézier curve to parameterize the orientation trajectory.

We now consider both the initial and lift-off trunk orientations $\mathbf{\Phi}_0$, $\mathbf{\Phi}_{lo}$, and the initial and lift-off trunk angular velocities $\dot{\mathbf{\Phi}}_0$, $\dot{\mathbf{\Phi}}_{lo}$. Setting these as boundary conditions, we obtain the following system, similar to (2.39):

$$\begin{cases} \mathbf{Q}_0 = \mathbf{\Phi}_0 \\ \mathbf{Q}_3 = \mathbf{\Phi}_{lo} \\ \mathbf{Q'}_0 = \dfrac{3}{T_{th}}(\mathbf{Q}_1 - \mathbf{Q}_0) = \dot{\mathbf{\Phi}}_0 \\ \mathbf{Q'}_1 = \dfrac{3}{T_{th}}(\mathbf{Q}_2 - \mathbf{Q}_1) \\ \mathbf{Q'}_2 = \dfrac{3}{T_{th}}(\mathbf{Q}_3 - \mathbf{Q}_2) = \dot{\mathbf{\Phi}}_{lo} \end{cases} \tag{3.9}$$

Here, $\mathbf{Q}$ is used to identify the control points in the angular Bézier curve, distinguishing them from $\mathbf{P}$, which is already used for the position Bézier curve.
Like the position trajectory, $\mathbf{Q}_1$ and $\mathbf{Q}_2$ must be determined. However, since the steps to find the solution are identical, we can express all the control points for the orientation and angular velocity trajectories as follows:

Orientation	Angular Velocity
$\mathbf{Q}_0 = \mathbf{\Phi}_0$	$\mathbf{Q'}_0 = \dot{\mathbf{\Phi}}_0$
$\mathbf{Q}_1 = \frac{T_{th}}{3}\dot{\mathbf{\Phi}}_0 + \mathbf{\Phi}_0$	$\mathbf{Q'}_1 = \frac{3}{T_{th}}(\mathbf{Q}_2 - \mathbf{Q}_1)$
$\mathbf{Q}_2 = -\frac{T_{th}}{3}\dot{\mathbf{\Phi}}_{lo} + \mathbf{\Phi}_{lo}$	$\mathbf{Q'}_2 = \dot{\mathbf{\Phi}}_{lo}$
$\mathbf{Q}_3 = \mathbf{\Phi}_{lo}$	

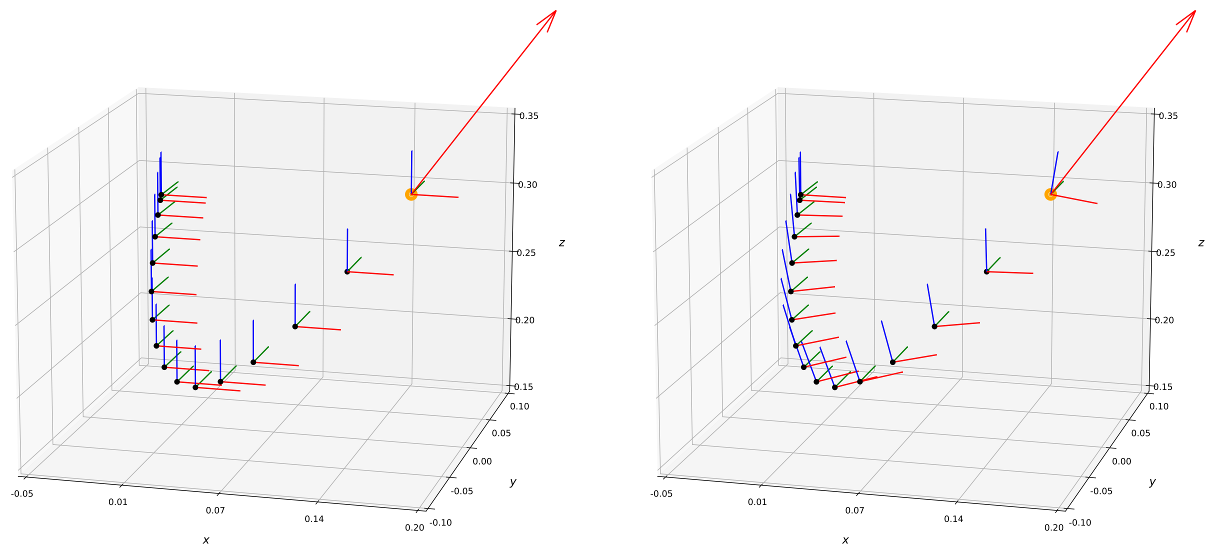
Table 3.2: Control point values for the two Bézier curves used in the thrust trajectory parametrization for the angular motion

The evaluation of the orientation trajectory is carried out as follows:

$$\text{Orientation-Bézier}(t) = \begin{cases} \mathbf{\Phi}(t) = \mathbf{B}(\frac{t}{T_{th}}, \mathbf{Q}) \\ \dot{\mathbf{\Phi}}(t) = \dot{\mathbf{B}}(\frac{t}{T_{th}}, \mathbf{Q}) \end{cases} \tag{3.10}$$

As mentioned, the only shared variable between the orientation and position trajectories is the thrusting time T_{th}, due to the duration constraints that both parameterizations must respect. Beyond this, the two trajectories remain completely independent, allowing for the execution of previously mentioned aerial maneuvers.

In Fig. 3.7, the 3D visualization of a thrust trajectory for a simple forward jump is shown, both for the pure position parameterization and for the combined position and orientation parameterization, with a pitch angular velocity of 4 $\frac{rad}{s}$. By comparing the two plots, we observe how incorporating the angular component could help distribute the GRFs through orientation changes and prepares for landing during the flight phase regulating the angular velocity at lift-off, causing the robot to pitch.



(a) pure position trajectory

(b) position and orientation trajectory

Figure 3.7: 3D visualization of the thrust trajectory with and without angular motion

3.3 Explosive Thrust

From our experiments and the results from the monopod work, we have confirmed that the thrust trajectory parameterization described so far works well. However, as previously explained, in robots, the compression phase of the thrust serves primarily to increase the range of leg extension that can be used for CoM acceleration, rather than to store potential energy (this applies to robots without spring-loaded or elastic actuators). This characteristic suggests that, intuitively, the decompression or pushing phase following the retracted configuration plays the most critical role in the thrust process.

In the current thrust trajectory parameterization, which is based purely on Bézier curves, the final lift-off velocity $\dot{\mathbf{c}}_{lo}$ and the thrust time T_{th} primarily influence how stretched the curve becomes. According to the control point calculations in Table 2.1, for a given fixed thrust time, if we increase the lift-off velocity while keeping other boundary conditions unchanged, control point $\mathbf{P}_2$ stretches the

curve, resulting in a longer path to travel within the same timeframe.
We realize that this behavior of the Bézier-based parameterization might limit the expressiveness of the thrust, particularly during the explosive decompression phase. Certain lift-off velocities could be excluded from the feasible set because the resulting trajectory could violate the robot's workspace constraints (i.g. the robot squats too much), such as requiring a lower height than feasible (i.e. the belly touches the ground), or even a negative value (which would be nonsensical, as the robot is standing on the ground).

By analyzing the trajectory generated in early training, we noticed that after the initial decompression stage, the path tends to flatten into a straight line. This observation suggests that splitting the trajectory parameterization could offer a solution. The idea is to split the thrust trajectory into two parts, while keeping the same control strategy, but modifying how the position and velocity references are calculated. This would allow us to maintain control over the thrust while ensuring that high lift-off velocities do not result in workspace violations or unfeasible heights.

Since the trajectory flattens into a straight line only toward the end, our idea is to maintain the same Bézier parameterization for the initial phase, while substituting the final part with a model based on the Uniformly Accelerated Rectilinear Motion (UARM).
This new parameterization should introduce as few additional parameters to the action space as possible, in order to keep the exploration space minimal.
Before determining how to select these parameters and how to connect the initial Bézier parameterization with this new model, let's first outline the UARM equations of motion:

$$\begin{cases} v_f = v_0 + a(t - t_0) \\ d_f = \frac{1}{2}a(t - t_0)^2 + v_0(t - t_0) + d_0 \end{cases} \overset{t_0=0}{\rightarrow} \begin{cases} v_f = v_0 + at \\ d_f = \frac{1}{2}at^2 + v_0 t + d_0 \end{cases} \tag{3.11}$$

Where v_0 and v_f are the initial and final velocities, d_0 and d_f are the initial and final displacements, and t_0 is the initial time. In our case, we set the initial time $t_0 = 0$, simplifying the system.
We are now interested in solving for the time variable to better understand its relationship with the other variables:

$$t = \frac{v_f - v_0}{a} \tag{3.12}$$

Substituting this expression for t back into the position equation, our goal is to isolate the acceleration term a:

$$\begin{aligned} d_f &= \frac{1}{2}a\left(\frac{v_f - v_0}{a}\right)^2 + v_0\left(\frac{v_f - v_0}{a}\right) + d_0 \\ d_f &= \frac{1}{2}\frac{(v_f - v_0)^2}{a} + \frac{v_0 v_f - v_0^2}{a} + d_0 \\ d_f - d_0 &= \frac{(v_f - v_0)^2 + 2(v_0 v_f - v_0^2)}{2a} \\ 2a(d_f - d_0) &= v_f^2 - v_0^2 \end{aligned} \tag{3.13}$$

Bringing everything together, we obtain the following system, which defines the time term t and the acceleration term a:

$$\begin{cases} a = \frac{1}{2}\frac{v_f^2 - v_0^2}{d_f - d_0} \\ t = \frac{v_f - v_0}{a} \end{cases} \tag{3.14}$$

Previously, our thrust trajectory was characterized by the initial and lift-off positions and linear velocities $(\mathbf{c}_0, \dot{\mathbf{c}}_0)$, $(\mathbf{c}_{lo}, \dot{\mathbf{c}}_{lo})$. Now, we introduce an intermediate configuration, $(\mathbf{c}_{lo_b}, \dot{\mathbf{c}}_{lo_b})$, which represents the endpoint of the Bézier parameterization and the starting point for the Uniformly Accelerated

Rectilinear Motion (UARM) trajectory. The actual lift-off configuration, which is the final condition of the UARM trajectory, is denoted as $(\mathbf{c}_{lo_e}, \dot{\mathbf{c}}_{lo_e})$.

Given this new concept and notation, the system 3.14 becomes:

$$\begin{cases} a = \dfrac{1}{2}\dfrac{\dot{\mathbf{c}}_{lo_e}^2 - \dot{\mathbf{c}}_{lo_b}^2}{\mathbf{c}_{lo_e} - \mathbf{c}_{lo_b}} \\ T_{th_e} = \dfrac{\dot{\mathbf{c}}_{lo_e} - \dot{\mathbf{c}}_{lo_b}}{a} \end{cases} \tag{3.15}$$

Where T_{th_e} is the thrust time under the UARM trajectory. We can now rename the thrust time under the Bézier curve as T_{th_b}, so the total thrust time becomes:

$$T_{th} = T_{th_b} + T_{th_e} \tag{3.16}$$

Introducing the explosive lift-off configuration into the action space would involve adding six new variables, as $\dot{\mathbf{c}}_{lo_e} \in \mathbb{R}^3$ and $\mathbf{c}_{lo_e} \in \mathbb{R}^3$. However, such an increase would be inconveniently large. A more efficient approach is to leverage the properties of the Bézier curve to define these new values.

For instance, we can express the explosive lift-off velocity term $\dot{\mathbf{c}}_{lo_e}$ as a scaled version of the Bézier lift-off velocity vector $\dot{\mathbf{c}}_{lo_b}$. Additionally, we can express the total displacement of the explosive trajectory $\mathbf{c}_{lo_e}$ as an offset from the Bézier lift-off position $\mathbf{c}_{lo_b}$ in the direction of the Bézier lift-off velocity. We introduce the velocity multiplier term k, so the explosive lift-off velocity is defined as:

$$\dot{\mathbf{c}}_{lo_e} = k\dot{\mathbf{c}}_{lo_b} \quad k \geq 1 \tag{3.17}$$

To describe the displacement of the explosive trajectory from the Bézier lift-off position, we first compute the unit vector based on the Bézier lift-off velocity to establish the direction of motion:

$$\hat{\dot{\mathbf{c}}}_{lo_b} = \frac{\dot{\mathbf{c}}_{lo_b}}{\|\dot{\mathbf{c}}_{lo_b}\|} \tag{3.18}$$

Next, we introduce the displacement value d, allowing us to define the explosive lift-off position as:

$$\mathbf{c}_{lo_e} = \mathbf{c}_{lo_b} + d\hat{\dot{\mathbf{c}}}_{lo_b} \quad d \geq 0 \tag{3.19}$$

By using this approach, instead of introducing six new parameters, only the two k and d are needed, adding enough expressiveness without increasing too much the dimensionality of the problem. This formulation ensures also that the explosive part is a direct continuation of the Bézier trajectory.
In Fig. 3.8, the new parameterization of the position and linear velocity is illustrated.

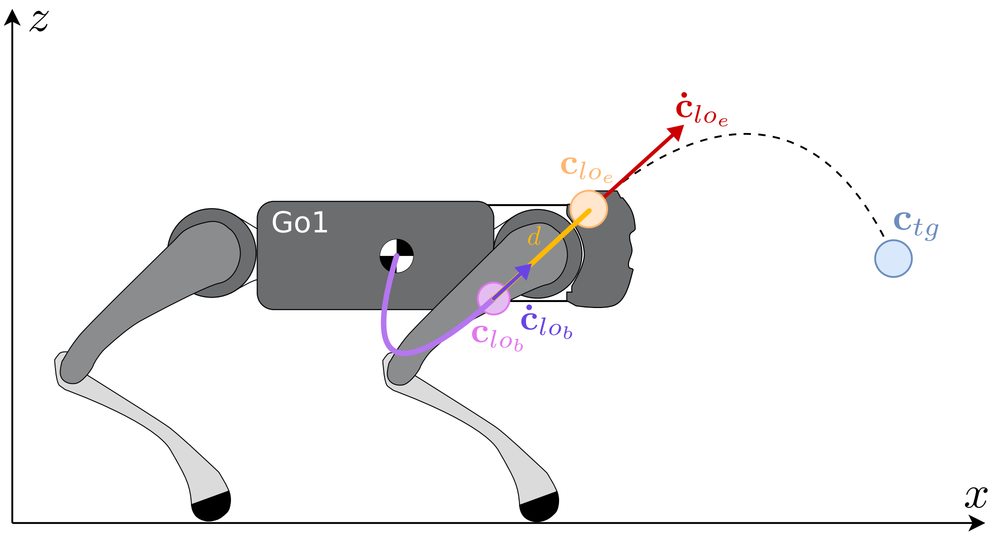


Figure 3.8: Thrust trajectory with the new Bézier-UARM parametrization

It is important to note that, despite this new parameterization, the policy can choose to exclude the contribution of the UARM phase for certain jumps by simply setting $d = 0$. In this case, only the Bézier trajectory will be used.

The evaluation of the UARM trajectory can be sped up with respect to evaluate the motion equation using a linear interpolation function, given the weight w, normalized such that $0 \leq w \leq 1$:

$$\text{lerp}(x_s, x_e, w) = x_s + w(x_e - x_s) \tag{3.20}$$

Thus, the UARM trajectory can be defined as:

$$\text{UARM}(t) = \begin{cases} \mathbf{c}(t) = \text{lerp}\left(\mathbf{c}_{lo_b}, \mathbf{c}_{lo_e}, \dfrac{t - T_{th_b}}{T_{th_e}}\right) \\ \dot{\mathbf{c}}(t) = \text{lerp}\left(\dot{\mathbf{c}}_{lo_b}, \dot{\mathbf{c}}_{lo_e}, \dfrac{t - T_{th_b}}{T_{th_e}}\right) \end{cases} \tag{3.21}$$

For reference, the Bézier position trajectory is expressed as:

$$\text{Position-Bézier}(t) = \begin{cases} \mathbf{c}(t) = \mathbf{B}\left(\dfrac{t}{T_{th_b}}, \mathbf{P}\right) \\ \dot{\mathbf{c}}(t) = \dot{\mathbf{B}}\left(\dfrac{t}{T_{th_b}}, \mathbf{P}\right) \end{cases} \tag{3.22}$$

where the control points, using the new notation, are:

Position	Velocity
$\mathbf{P}_0 = \mathbf{c}_0$	$\mathbf{P'}_0 = \dot{\mathbf{c}}_0$
$\mathbf{P}_1 = \frac{T_{th_b}}{3}\dot{\mathbf{c}}_0 + \mathbf{c}_0$	$\mathbf{P'}_1 = \frac{3}{T_{th_b}}(\mathbf{P}_2 - \mathbf{P}_1)$
$\mathbf{P}_2 = -\frac{T_{th_b}}{3}\dot{\mathbf{c}}_{lo_b} + \mathbf{c}_{lo_b}$	$\mathbf{P'}_2 = \dot{\mathbf{c}}_{lo_b}$
$\mathbf{P}_3 = \mathbf{c}_{lo_b}$	

Table 3.3: Control point values for the two Bézier curves used in thrust trajectory parametrization with the new notation

In Fig. 3.9, the two resulting trajectories are shown: on the left, the new parameterization method, and on the right, the previous pure Bézier one. In this example, a high final lift-off velocity with a magnitude of 3 $\frac{m}{s}$ is required in both cases, with the same thrust time T_{th}. As illustrated, the previous parameterization produces an unfeasible trajectory that violates the minimum height constraint of $z \geq 0.15\,\mathrm{m}$. This demonstrates that the new parameterization can generate explosive thrust trajectories while satisfying the system's feasibility constraints.

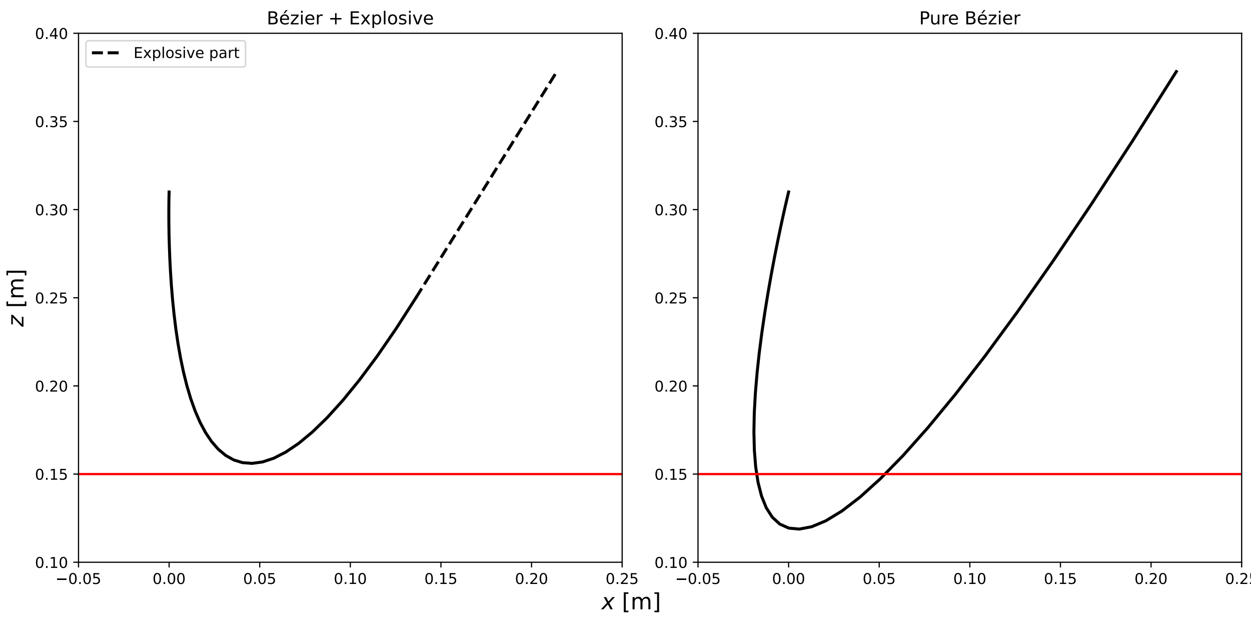


Figure 3.9: Comparison of the two thrust trajectories computed using the new parameterization and the previous pure Bézier method

In the end, the complete thrust trajectory is evaluated as follows:

$$\begin{aligned} \mathbf{c}(t), \dot{\mathbf{c}}(t) &= \begin{cases} \text{Position-Bézier}(t) & 0 \leq t \leq T_{th_b} \\ \text{UARM}(t) & T_{th_b} < t \leq T_{th_e} \end{cases} \\ \mathbf{\Phi}(t), \dot{\mathbf{\Phi}}(t) &= \text{Orientation-Bézier}(t) \qquad 0 \leq t \leq T_{th} \end{aligned} \tag{3.23}$$

This formulation allows for the combination of both the Bézier curve-based trajectory during the initial thrust phase, which covers the decompression stage, and the UARM-based model for the following, more explosive part of the trajectory. The switch between these two phases occurs at time T_{th_b}, ensuring a smooth transition from the initial to the final phase of the thrust (by construction). Meanwhile, the orientation trajectory is independently governed by the Bézier parameterization, maintaining consistency across the entire thrust phase. This setup provides flexibility for both thrust and orientation control, enabling complex and dynamic maneuvers to be achieved.

3.4 The new Action and State Spaces

The design of both the state and action space plays a crucial role in policy learning and in ensuring the general applicability of the learning framework, or the learned policy, across multiple robots with different morphologies and characteristics. In our previous work, both spaces were designed in such a way that no information from the joint space was included. Specifically, the state space contained only the robot's CoM position (expressed in world frame) and the target position. The action space, as defined in Eqs. 3.1 and 3.3, included thrust time, while the remaining terms were expressed directly in the task space.

This task-space-centric approach allows for independence from the morphology of specific robotic platform, meaning the same policy can be applied to different robots regardless of variations in their joint configurations or DoFs. For example, if joint states were incorporated, the action space would need to adapt to each robot's specific leg configuration, leading to a more complex and less generalizable policy. This concept is emphasized and further elevated in the work of Shafiee et al. [33], where a single locomotion policy was successfully designed and applied across 16 quadrupeds with varying masses, heights, and morphologies.
The key idea here is to abstract away the details of the joint space and leave the mapping from task space to joint space to the inverse kinematics (IK). This strategy leverages the robustness of low-level controllers to ensure accurate tracking performance, enabling reusable policies across different robots.

With this in mind, we will now define our design choices for the new state space, which simplifies the jump command description, and the action space, which incorporates the thrust trajectory parameterization developed so far.

3.4.1 State Space

As described in the RL key concepts subsection 2.3.1, the state s_t is a representation of the environment at a specific time t. This state is fundamental for the policy to determine the appropriate action to perform. In our previous monopod work, the state space was composed of the current location and the desired target location for the jump (since the angular part was neglected):

$$\mathbf{s}_{\text{old}} = (\mathbf{c}, \mathbf{c}_{tg}) \in \mathbb{R}^6 \tag{3.24}$$

Notably, this state includes not only the robot's actual position but also what can be seen as a command, in the form of the target location, which influences the action decision. While this formulation is simple, it presents a challenge for real-world applicability, specifically regarding accurate state estimation of the robot's position.
A potential solution to this issue is to assume the robot is on flat ground, use forward kinematics FK to retrieve its initial configuration, and then express the target as a displacement from this value. However, we realized that this formulation is inefficient. Since the target can be directly expressed as a displacement, it makes sense to simplify the state space accordingly.
In our scenario, where the robot always starts from a default joint configuration (consequentially the initial position does not change), the initial configuration of the robot is redundant.
Thus, the new state space is more efficiently formulated as:

$$\mathbf{s} = (\Delta\mathbf{c}, \Delta\mathbf{\Phi}) \in \mathbb{R}^6 \tag{3.25}$$

This new state space directly captures the displacement in both position and orientation, streamlining the policy's decision-making process.
Although this formulation meets the requirements of our current work, a potential extension for future research, specifically for quadrupeds, could involve integrating the current linear and angular velocities obtained through IMU processing. Additionally, incorporating the position of the feet relative to the base frame could provide valuable information. This extended state representation would improve the

policy capability to better perform jumps from a wider range of conditions, such as inclined planes or non-stationary initial states, further enhancing the robot's adaptability and performance in diverse environments.

3.4.2 Action Space

So far, we have discussed how the parameters of the thrust trajectory are the values that the policy must predict based on the input state. In reference to the monopod work, the previous action (Eq. 3.3) consisted of thrust time T_{th}, along with the simplified polar spherical representation of both the lift-off position $\mathbf{c}_{lo}$ and linear velocity $\dot{\mathbf{c}}_{lo}$.
In the new thrust trajectory parameterization introduced in this thesis (Eq. 3.23), the following parameters are required as inputs to compute the trajectory:

- T_{th_b}: the total time of thrust for which the initial Position-Bézier is used
- $\mathbf{c}_{lo_b}$: the lift-off position used for the Position-Bézier trajectory
- $\dot{\mathbf{c}}_{lo_b}$: the lift-off velocity used for the Position-Bézier trajectory
- $\mathbf{c}_{lo_e}$: the lift-off position used for the UARM trajectory, derived using the displacement term d
- $\dot{\mathbf{c}}_{lo_e}$: the lift-off velocity used for the UARM trajectory, derived using the velocity multiplier term k
- $\mathbf{\Phi}_{lo}$: the lift-off orientation used for the Orientation-Bézier trajectory
- $\dot{\mathbf{\Phi}}_{lo}$: the lift-off angular velocity used for the Orientation-Bézier trajectory

It is important to note that, for the Orientation-Bézier trajectory, the total thrust time T_{th} is obtained by summing T_{th_b} and T_{th_e} (Eq. 3.16), where the latter is computed analytically (Eq. 3.15).

To reduce the dimensionality of the Position-Bézier lift-off configuration, we can apply the same simplification used in the previous monopod work, where these values are expressed in spherical coordinates (Eq. 3.2). This approach can also be used here, as illustrated in Fig. 3.10.

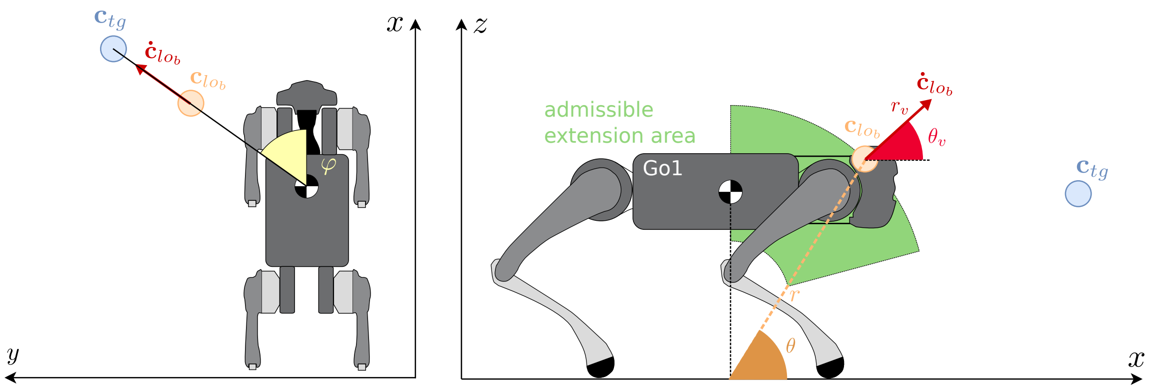


Figure 3.10: Top and side views of the action in spherical coordinates for the Go1 robot

Given this, we can now define the new action space, which encompasses both the position and angular components of the thrust trajectory, as follows:

$$\begin{aligned}\mathbf{a} &= (T_{th_b}, \mathbf{c}_{lo_b}, \dot{\mathbf{c}}_{lo_b}, \mathbf{\Phi}_{lo}, \dot{\mathbf{\Phi}}_{lo}, k, d) \in \mathbb{R}^{15} \\ &= (T_{th_b}, r, \theta, r_v, \theta_v, \mathbf{\Phi}_{lo,\psi}, \mathbf{\Phi}_{lo,\theta}, \mathbf{\Phi}_{lo,\varphi}, \dot{\mathbf{\Phi}}_{lo,\psi}, \dot{\mathbf{\Phi}}_{lo,\theta}, \dot{\mathbf{\Phi}}_{lo,\varphi}, k, d) \in \mathbb{R}^{13}\end{aligned} \tag{3.26}$$

As you can see, the introduction of the angular component has more than doubled the number of parameters (w.r.t eq. 3.3), while the splitting of the position trajectory has increased the overall

number of values by only two, as initially planned. Similar to the previous monopod work, the ranges of these values can be constrained using domain knowledge to further narrow the search space.
For conciseness, the values, their respective ranges, and a brief explanation are provided below:

- T_{th_b} [0.4, 1.0]: The duration of the thrust phase where the Bezier curve governs the position trajectory. This phase must be executed rapidly to achieve a controlled lift-off, while allowing enough time for precise thrust generation.
- r [0.15, 0.4]: The extension radius of the robot's legs during the thrust phase. This range prevents boundary singularities that could arise from overextension or complete retraction of the legs, ensuring a feasible configuration throughout the Position-Bézier thrust phase.
- θ [$\frac{\pi}{4}$, $\frac{\pi}{2}$]: The pitch angle of the robot's body, responsible for balancing the vertical and horizontal thrust components. This range prevents excessive foot slippage and avoids unnecessary force exertion during the Position-Bézier thrust phase.
- r_v [0.5, 5]: The magnitude of the Position-Bézier lift-off velocity. This parameter influences how far the robot will jump, but it is constrained by an upper bound to limit policy exploration during the search for an optimal value that remains physically feasible for the system.
- θ_v [$\frac{\pi}{6}$, $\frac{\pi}{2}$]: The pitch angle of the Position-Bézier velocity vector at lift-off, which regulates the balance between horizontal and vertical velocity contributions, impacting the jump trajectory.
- $\mathbf{\Phi}_{lo,\psi}$ [$-\frac{\pi}{6}$, $\frac{\pi}{6}$]: The roll angle at lift-off, allowing for small adjustments to stabilize the robot's orientation during take-off, particularly when performing jumps on uneven terrain.
- $\mathbf{\Phi}_{lo,\theta}$ [$-\frac{\pi}{6}$, $\frac{\pi}{6}$]: The pitch angle at lift-off, allowing for minor adjustments to the robot's body orientation, which helps maintain balance and optimize the thrust force distribution.
- $\mathbf{\Phi}_{lo,\varphi}$ [$-\frac{\pi}{4}$, $\frac{\pi}{4}$]: The yaw angle at lift-off, contributing to rotational maneuvers such as twist jumps. This parameter enhances the flexibility of the robot's orientation control during more dynamic movements.
- $\dot{\mathbf{\Phi}}_{lo,\psi}$ [-1, 1]: The angular velocity around the roll axis, allowing for minor roll adjustments during the airborne phase, essential for maintaining stability in mid-air.
- $\dot{\mathbf{\Phi}}_{lo,\theta}$ [-1, 1]: The angular velocity around the pitch axis, enabling slight pitch adjustments during the flight phase to improve landing posture.
- $\dot{\mathbf{\Phi}}_{lo,\varphi}$ [-4, 3]: The angular velocity around the yaw axis, which contributes to generating sufficient rotation for twist jumps, allowing the robot to reorient itself mid-air.
- k [1, 3]: The velocity multiplier for the explosive thrust phase. This term scales the Bezier lift-off velocity to create a more explosive jump when necessary, though excessively high values could make the trajectory untrackable by the controller.
- d [0, 0.3]: The displacement during the explosive thrust phase, regulating how much additional distance the robot travels after the Bezier phase. This also impacts the duration of the final part of the thrust phase.

A small clarification from an implementation perspective is necessary. Unlike in TD3, where the action produced by the Actor NN is passed through a tanh layer and consequentially bounded to a range of $[-1, 1]$, the PPO algorithm computes actions by sampling from a normal distribution $a \sim \mathcal{N}(\mu, \sigma^2)$, which is not inherently constrained.
To prevent actions from exceeding the defined range, clipping is introduced. Although this operation does not compromise the mathematical formulation of PPO, the choice of clipping range significantly impacts policy performance [61]. Therefore, determining the appropriate clipping range for PPO becomes an important hyperparameter, which we have set to $[-5, 5]$. The handling of this mechanism is further discussed in the reward section.

3.5 Reward Function

As outlined in the previous work in subsection 3.1.2, the reward function serves as the sole indicator that evaluates the effectiveness of an action in achieving the task goal. In the previous work, the reward function was formulated as a summation of penalties subtracted from a primary positive reward term that assessed the task's main objective: how closely the robot landed to the target location after jumping. Moreover, the reward function can act as a powerful tool to inject engineering principles and domain-specific knowledge into the learning process.

The core idea behind our reward design is that achieving the maximum reward related to the task goal requires adherence to and minimization of a set of physical system constraints. We have previously introduced the concept of path constraints, but to recap: a path constraint is a restriction that must be satisfied throughout the entire thrust phase. Any violation of these constraints results in a penalty. These path constraints are evaluated at each time step, and violations are penalized using the linear activation function $A(x, \underline{x}, \overline{x})$ defined in equation 3.4. All penalties are accumulated, as in the previous work, into a feasibility cost C_f.

The set of path constraints, essential for ensuring the physical feasibility and safe execution of the jump, remains consistent with the monopod work but is now adapted to the quadruped platform. These constraints are evaluated at every time step and include: joint position, velocity, and torque limits, friction cone, unilaterality, and stay away from singularity. By minimizing the penalties associated with any violations, the policy ensures that the robot operates in a physically feasible and reliable manner throughout the entire jump.

As observed in the previous work, while path constraints help ensure physical feasibility, they are not sufficient to fully guide the policy towards optimal jump behavior. Additional costs are needed to promote desirable behaviors while penalizing unwanted actions. These costs differ from path constraints as they focus more on guiding specific behavior during the jump. Some previous costs have been revised or removed, while new ones have been introduced.
The updated constraint costs are as follows:

- **Lift-off tracking error** (C_{lo}): This is the only cost retained from the previous work, but it has been expanded. Previously, it only covered the linear part of the trajectory; now, it includes both linear and angular components. This cost ensures that the trajectory performed by the robot is accurately tracked by the low-level controller, ensuring feasibility. For the linear part, the final lift-off configuration is defined by the UARM parametrization.

- **Target orientation error** ($C_{\mathbf{\Phi}_{tg}}$): Instead of incorporating this as part of the main positive reward, we treat it as a penalty that functions as a tracking error. While the primary goal is to reach the target location, minimizing the orientation error at landing is crucial for task success.

- **Touchdown bounce penalization** ($C_{\Delta x}$): This regularization cost addresses landing instability caused by the simplified landing control. Excessive horizontal velocity during touchdown may lead to bouncing behavior, which can destabilize the robot which can tip over. In the previous work on monopod, there was not this need since the episode was ended once the touchdown was detected. This cost replaces the "no touchdown" penalty from the monopod work, assigning a high default value, the cost is recalculated based on the distance between the final position at episode timeout and the touchdown location if detected.

- **Touchdown angular velocity penalization** ($C_{\dot{\mathbf{\Phi}}_{td}}$): This penalty discourages high angular velocities at touchdown, which could cause the robot to lose balance or suffer mechanical damage, leading to a failed jump.

- **Action limit penalization** (C_{ppo}): As mentioned earlier, PPO samples actions from a Normal distribution, which is not naturally bounded. To avoid excessive actions, clipping is introduced.

This penalty informs the policy when an action exceeds the predefined bounds, thereby helping the policy stay within the safe and feasible range.

These additional costs guide the policy towards safer, more stable, and physically feasible jump execution, further refining the policy's behavior during training.

Now we can address the main positive reward term introduced earlier: the landing target reward function (R_{lt}). This term measures how close the final landing location (evaluated at the episode's timeout) is to the target location. While the previous definition (Eq. 3.5) was sufficient, we wanted to enhance it by incorporating the fact that errors must be weighted differently depending on the jumping magnitude. The goal is to better distinguish the impact of landing errors err_{tf} based on the jump distance. For example, an error of 0.2 meters is significant for a 0.1 meter long jump but less severe for a 1 meter jump.
The revised landing target reward function is defined as follows:

$$\text{err}_{tg} = \|\mathbf{c} - \mathbf{c}_{tg}\| \quad \Delta_{tg} = \|\mathbf{c}_{tg} - \mathbf{c}_0\| \tag{3.27}$$

$$R_{lt}(\mathbf{c}, \mathbf{c_0}, \mathbf{c}_{tg}) = 1_{\mathbb{R}+} \left[\left(\log \left(1 + \frac{1}{\alpha \ \text{err}_{tg} \ + \epsilon} \right) + \beta e^{\Delta_{tg}} \right) (1 - \text{err}_{tg})^{\gamma} - b \right] \tag{3.28}$$

where:

- α is a hyperparameter regulating the influence of the landing error,
- β controls the contribution based on the jump distance,
- γ adjusts the weight of the landing error penalty,
- ϵ prevents division by zero,
- b is a bias term that regulates the reward function's overall shape and height.

This formulation enhances the reward function by making it sensitive to the magnitude of the jump, the landing error and the magnitude of the jump, ensuring that the policy is encouraged to perform well across different jump distances.
In Fig. 3.11, the surface plot of this reward function is depicted for realistic jump distances within the range $[0, 1]$ meters and landing errors within $[0.001, 0.5]$ meters, using the following hyperparameter values: $\alpha = 0.5$, $\beta = 5$, $\gamma = 4$, and $b = 1$. As can be observed, the reward increases as the landing error approaches zero. Furthermore, for the same landing error, the reward value increases proportionally with the desired jump distance, reflecting the relative importance of accuracy across different distances.

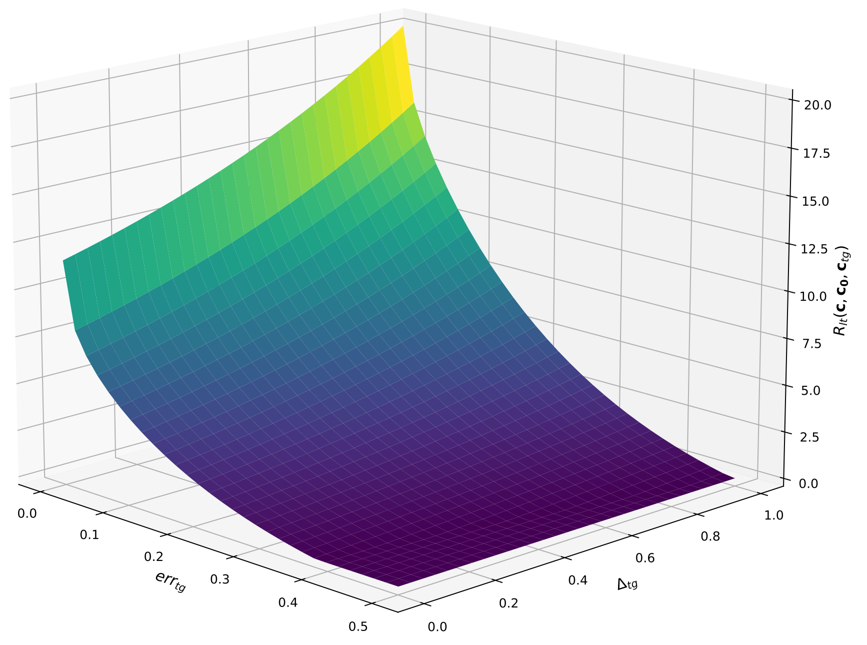


Figure 3.11: Landing target reward function surface

Now that all the components of the reward function have been defined, we can construct the total reward function. In the monopod work, the reward was structured as the landing target reward minus the sum of all constraint costs, clamped then to the positive part. While this formulation worked well, the clamping might have removed valuable information for the policy during the learning process. Inspired by the reward formulation of Atanassov et al. [37], we introduce a more refined reward function:

$$R = R_{lt}(\mathbf{c}, \mathbf{c_0}, \mathbf{c}_{tg})e^{-(\sum C_i)^2} \tag{3.29}$$

In this new formulation, the reward is always positive and dynamically scaled according to the penalty costs. When the sum of penalties approaches infinity, the negative sign in the exponent forces the reward towards zero, effectively nullifying the positive reward contribution. Conversely, when the penalty cost sum approaches zero, the exponential term tends towards one, preserving the full value of the landing target reward. This approach ensures that the reward function remains informative throughout the learning process, promoting better convergence and policy performance.

In conclusion, to maximize the reward, the policy must effectively minimize the cost penalties, thereby ensuring that all system constraints are respected. This approach guarantees that the proposed trajectory is both feasible and trackable, allowing the robot to perform optimal jumps while adhering to the physical limitations of the system. This structure promotes not only task completion but also the safe and efficient execution of the learned behaviors in real-world scenarios.

4 Results

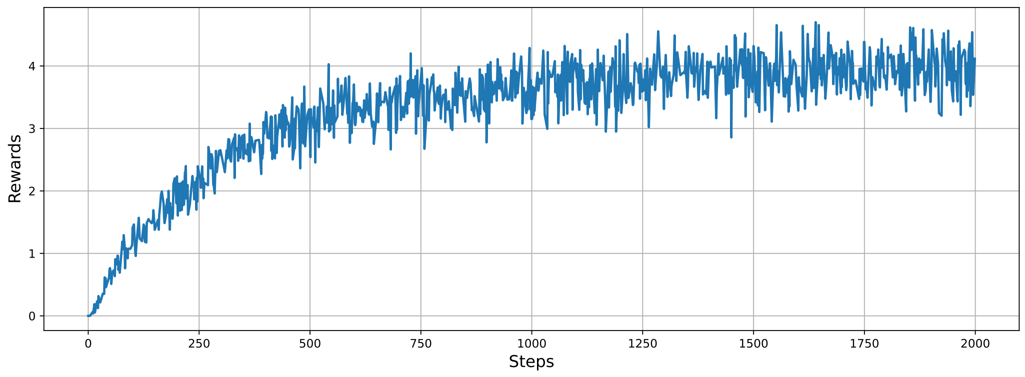
In this section, we will evaluate and present the results achieved in this work through a simulation-based validation of the learned policy. Before diving into the detailed analysis, it is essential to outline the specifications of the hardware used for both training and evaluation, as these significantly impact training time and parallelization capabilities. The simulations were executed on a desktop equipped with an Intel I7-13700K 24-threads CPU, 16 GB of DDR4 RAM, and an Nvidia RTX 3060 GPU with 16 GB of VRAM, running on Ubuntu 22.04 LTS. The policy training exploited parallelization capabilities by simulating 4096 robots simultaneously within Isaac Sim.
For each completed episode, one PPO training step was performed, amounting to a total of 2000 training steps. Training the policy under this setup took approximately 15 hours. However, it's important to emphasize the efficiency of our approach when compared to typical E2E methods, such as those available in the Orbit framework. These methods generally require 20 to 25 episode steps of recorded data to complete a single training step. In contrast, our method performs a training step with just one episode step, corresponding to the entirety of the episode since the episode is composed of only one action. This means that, given an equivalent number of policy training steps, our approach uses 20 times less data, highlighting the remarkable sample efficiency of the proposed framework.

We define the training region for the policy as a rectangular space with the X-axis ranging from -0.6 to 1.2 meters, the Y-axis from -0.6 to 0.6 meters, and the Z-axis allowing jumps up to 0.4 meters and down to -0.4 meters. This setup ensures coverage of both forward and backward jumps, providing a broad exploration of possible landing configurations.
For the angular component, the training region includes small roll and pitch perturbations, limited to a maximum of 15 degrees, ensuring the robot can handle minor tilts during the jump. However, the yaw angle, which is of greater importance in this work, is allowed to vary significantly, covering a range from -90 to 90 degrees. This range ensures the policy can perform jumps with diverse orientations, preparing the robot for more dynamic scenarios.
In Fig. 4.1, we present both the reward curve and the cumulative penalty sums for each training step. A crucial observation is the rapid convergence of cumulative penalties to a small value, suggesting that the learned policy quickly adapts to producing physically feasible actions and adheres to system constraints. This serves as strong evidence that our reward formulation is effective, driving the policy towards solutions that satisfy the constraints while optimizing the task's objectives.
Furthermore, we will delve into a variety of tests:

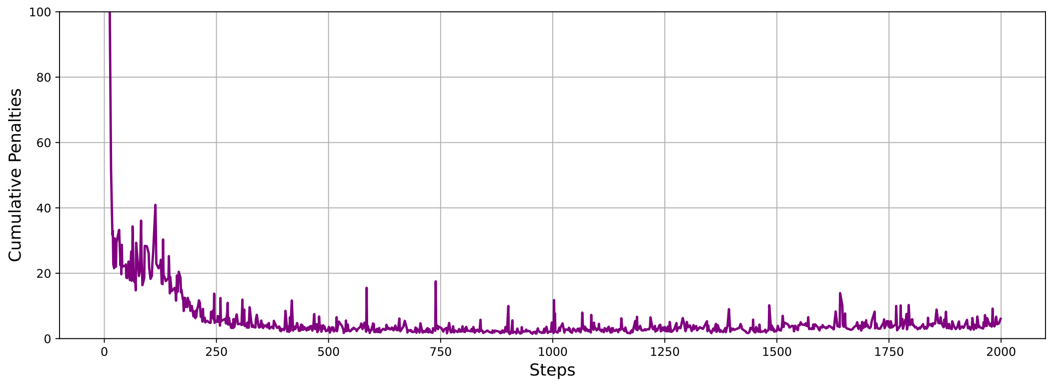
- **Linear Trajectory Testing**: This evaluates the learned policy's ability to handle jumps of varying heights (including downward jumps) without the incorporation of an orientation goal.
- **Orientation-Based Testing**: In this case, we assess in-place jumps, including pure yaw rotations, to demonstrate the policy's ability to manage jumps that involve dynamic orientation changes.

We will also analyze joint-level trajectories, including joint positions, velocities, and torques, to ensure that the proposed trajectories are feasible and adhere to the system's physical constraints.
Finally, we conduct a comparison between our RL-based approach and the state-of-the-art multi-stage E2E alternative proposed by Atanassov et al. [37], which was also implemented on the Unitree Go1 robotic platform. This method is currently regarded as the benchmark for RL-based jumping, offering a valuable reference for evaluating the performance of our approach.

The comparison will focus on key metrics such as jump distance and landing accuracy, providing a clear assessment of the advantages and limitations of each method.
Additionally, we demonstrate the generalization capability of our learned policy by applying it to a multi-platform scenario, specifically performing zero-shot transfer to the Aliengo robot. This experiment highlights the adaptability of our framework, showcasing how the policy can seamlessly transfer to similar robotic platforms that share common actuation systems and characteristics with the Go1.
Through these extensive tests and comparisons, we aim to emphasize the efficiency, adaptability, and robustness of our proposed RL-based jumping strategy.



(a) Reward progression over training episodes



(b) Summed penalties over training episodes

Figure 4.1: Training curves for reward (top) and summed penalties (bottom) over training episodes, highlighting the policy's improvement and its adherence to physical constraints

4.1 Validation of Omnidirectional Jumps

In this section, we evaluate the performance of the learned policy in executing omnidirectional jumps, a crucial skill for quadruped robots when navigating complex and unpredictable environments. The ability to jump in multiple directions, combined with the capability to adapt to varying surface heights, enhances the system's versatility for real-world scenarios, where simple, single-direction jumps or fixed-height landings are often insufficient.

We focus on testing the policy's robustness and adaptability to both horizontal and vertical displacements, as well as its effectiveness in controlling re-orientation during dynamic jumps. Specifically, the omnidirectional jump validation will encompass forward, backward, and lateral jumps, as well as those involving upward or downward motions on uneven surfaces. Additionally, we will assess rapid in-place jumps with twist rotations, demonstrating the robot's ability to execute complex maneuvers efficiently.

These tests are performed within the previously established training region, providing a comprehensive evaluation of the learned policy. Through this validation, we aim to showcase the policy's precision in reaching target positions and orientations, while also highlighting its computational efficiency (an essential factor for real-time performance in dynamic environments).

4.1.1 Flat Jumps and Feasible Region

The concept of the feasible region refers to the maximum area in which the robot can perform jumps while keeping the landing error below a specified threshold. For this evaluation, we conducted 8,000 jumps at various target positions, all with zero height displacement. The absolute landing error (the distance between the actual landing position and the intended target) was calculated for each sample. A threshold of 0.2 meters on the landing error was applied to classify each jump as acceptable or not.

As depicted in Fig. 4.2, the computed feasible region illustrates the performance of the robot across the tested positions. A key observation is the symmetry of performance along the x-axis, which reflects the symmetrical configuration of the robot's legs. This symmetry confirms that the robot achieves consistent jumping accuracy for lateral displacements, with no noticeable difference between rightward and leftward jumps. Although the jumps in the backward direction were not tested over long distances, the results suggest similar performance to frontward jumps.

The logarithmic color scale used in the figure emphasizes the subtle differences in landing precision, particularly as the jumping distance remains within a certain range. This approach enhances the perception of the robot's performance within the 0.5-meter radius from the origin, where it achieves exceptional accuracy. Landing errors in this range are consistently below 5 cm, showcasing the precision and effectiveness of the learned policy for short to medium-range jumps.

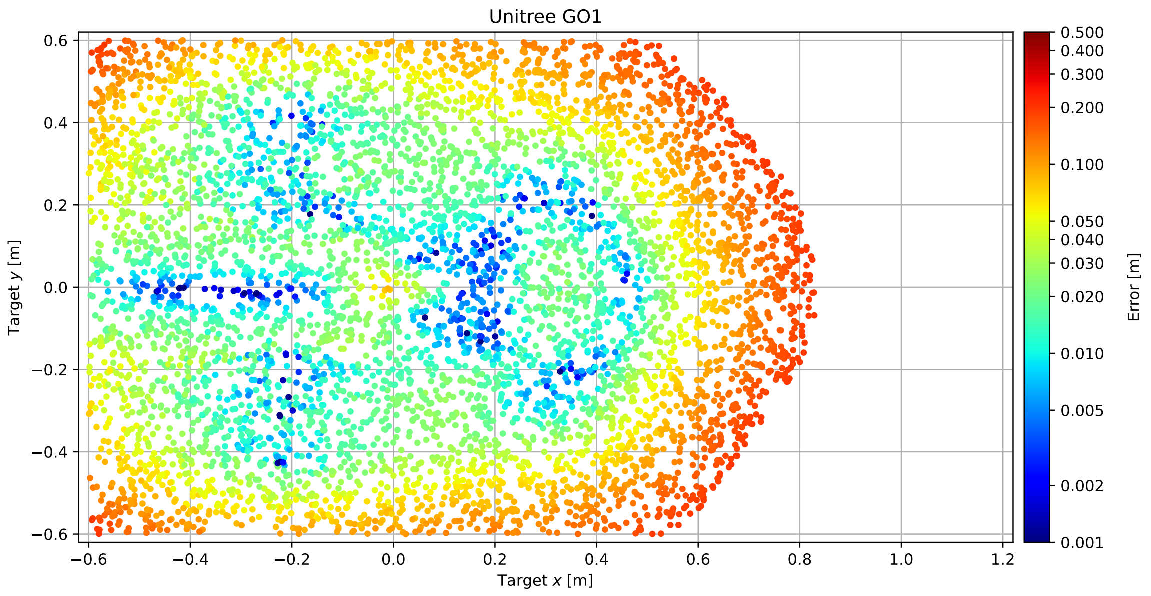


Figure 4.2: Feasible region for flat jump (no orientation)

When examining jumps with a target distance approaching 0.6 meters, it becomes evident that the control policy grows increasingly conservative. This conservatism is reflected in the robot's inability to execute larger jumps beyond this distance. To gain a clearer understanding of the robot's behavior during longer jumps, Fig. 4.3 presents a comparison between the target and actual distances achieved by the robot for both front and back jumps.

In an ideal scenario, the robot's actual jump distances would align perfectly with the desired target, depicted by the dashed diagonal line in the plot.

The green area in the figure represents the actual performance of the policy. As expected, for target distances up to 0.6 meters, the policy performs well, showing precise and consistent jumps. The relatively narrow distribution of points in this range indicates that the robot maintains good accuracy as the target distance varies within this limit.
However, as the target distance exceeds 0.6 meters, the policy exhibits more cautious behavior. The green shaded area corresponds to jumps executed without any collisions involving non-foot parts of the robot. As you can see, no failed jumps occurred during the entire test. This cautious approach indicates that the policy prioritizes maintaining feasibility and safety over maximizing jump distance, ensuring stable landings and reducing the risk of unsuccessful jumps. This behavior reflects a conservative strategy that focuses on producing reliable, feasible jumps rather than pushing the system to possibly dangerous physical limits. Implementing a NN dedicated to improving landing proficiency is expected to significantly enhance the current performance, particularly in terms of extending the jumping distance.

This behavior is consistent with the trend observed in the reward plot, where minimal violations of constraints were recorded. This indicates that the robot effectively tracks the desired trajectory, allowing for the implementation of physical feasibility checks (as discussed in the previous monopod work sections). These checks serve to abort any jumps that are unlikely to cover the desired distance, ensuring that the robot only attempts jumps within its feasible range. This mechanism enhances safety and reliability by preventing the execution of jumps that could lead to failures or unsafe outcomes.

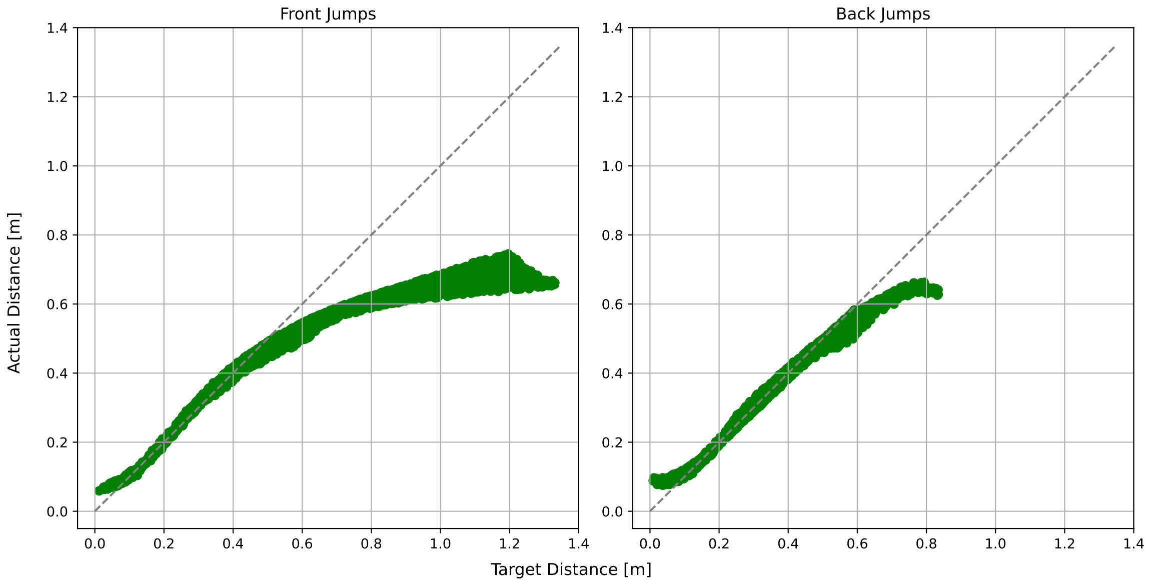


Figure 4.3: Target vs Actual Jump distance for both front and back region

4.1.2 Upward and Downward Jumps

In this subsection, we evaluate the performance of the learned policy in executing jumps with both positive and negative vertical displacements, focusing on upward and downward jumps. These types of jumps are critical for robotic navigation on uneven terrains or in environments with varying surface elevations, where the ability to adapt to different heights is essential for effective mobility.
The tests were conducted within the same horizontal positional region as before, but with target locations incorporating varying displacements along the z-axis. These ranged from upward jumps to higher platforms (positive vertical displacement) to downward jumps onto lower surfaces (negative vertical displacement). The goal of these tests was to assess how well the policy generalizes across different height levels and to understand the robot's limitations when dealing with height variations. The results of these tests were used to generate height maps for both upward and downward jumps. In the case of upward jumps, the maximum achievable height is recorded, while for downward jumps,

the minimum height is noted. These maps provide insight into the system's performance limits when combined with the constraints defined by the method.

In Fig. 4.4, the resulting height maps for both categories of jumps are presented. These maps were computed by uniformly sampling target points from the training region, and then filtering them using the same feasibility evaluation method described earlier. The feasibility check ensures that only physically achievable jumps are considered in the analysis. Clustering was applied to these filtered points to create a meshgrid, and each sample was assigned to a specific cluster. For each cluster, the maximum and minimum heights were calculated, allowing us to visualize the robot's capability in both positive and negative z-axis displacements.

As shown in the left contour plot, which reports the results for upward jumps, the robot is capable of executing jumps to surfaces with up to 0.26 meters of vertical displacement. The total range of successful upward jumps lies between 0.16 meters and 0.28 meters, with some variation depending on the lateral displacement. Notably, the robot tends to struggle with upward jumps targeting locations in the back region, particularly when combined with significant lateral displacement. This behavior is expected due to the kinematic constraints of the robot in these specific configurations, where the required force distribution for pushing off is less efficient, and the available range of motion becomes more limited.
In contrast, the right contour plot demonstrates that the robot can successfully perform downward jumps across its entire feasible region, with vertical displacements of up to -0.4 meters, which aligns with the bounds established during the training and test phase. This indicates that the robot can handle larger drops more effectively, as the gravitational force assists in achieving the target, making the kinematic demands less stringent compared to upward jumps.
These results demonstrate the versatility of the learned policy in handling both upward and downward jumps, showcasing its robustness and adaptability across different vertical displacements. The analysis highlights that while upward jumps are limited by the robot's kinematics, especially in backward or lateral directions, downward jumps are more easily achieved due to the assistance of gravity, allowing for greater vertical displacement. This confirms the efficacy of the proposed approach in real-world applications where navigating height variations is crucial.

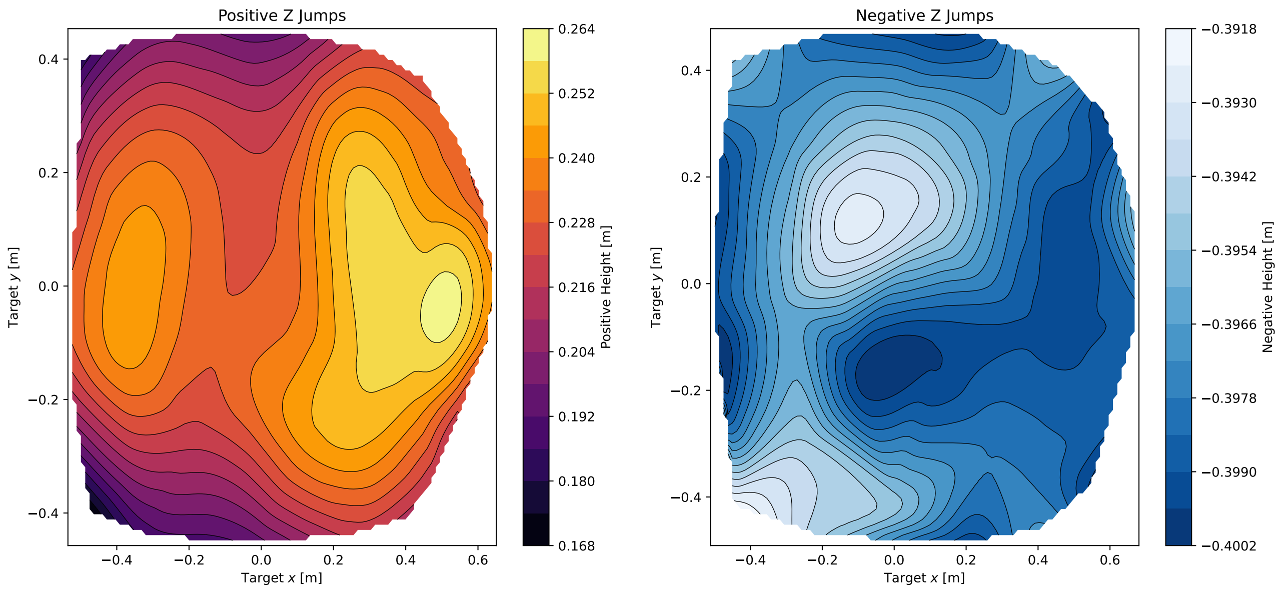


Figure 4.4: Height map for both upward jumps (on the left) and downward jumps (on the right)

4.1.3 Twist Jumps

Thus far, our evaluation has focused solely on jump performance without considering orientation changes in the target. However, incorporating orientation adjustments during jumps is crucial for enabling the robot to execute rapid reorientation maneuvers, improve the precision of lateral jumps, and prepare for subsequent locomotion strategies. The ability to control orientation mid-air significantly enhances the robot's versatility, particularly when faced with dynamic tasks requiring not only precise landing but also specific orientation for optimal post-jump actions.
In this test, we performed a series of in-place jumps where each jump involved a different target yaw orientation change. This scenario is particularly suited to demonstrate the system's ability to manage angular adjustments during the thrust phase. The test examines whether the learned policy can effectively control the robot's orientation while ensuring it reaches the desired landing position.

In Fig. 4.5, the landing orientation error relative to the desired yaw angle is shown. In-place jumps with no orientation change serve as a baseline, resulting in minimal orientation error, as expected. However, as the target yaw angle increases (both positively and negatively) the orientation error rises symmetrically. This symmetric behavior reflects the robot's consistent ability to handle yaw rotations in either direction.
The maximum observed yaw error is approximately 15 degrees, which is a satisfactory result, considering that it stays under 20% of the desired yaw angle across a wide range of rotations. This level of error indicates that the learned policy is proficient at managing dynamic jumps involving significant reorientation, maintaining a reasonable level of accuracy even under challenging angular displacement scenarios. This capability is particularly beneficial for executing maneuvers that require quick rotations, such as twist jumps or adjustments to prepare for subsequent locomotion, further validating the robustness and versatility of the proposed approach in managing both linear and angular trajectories.

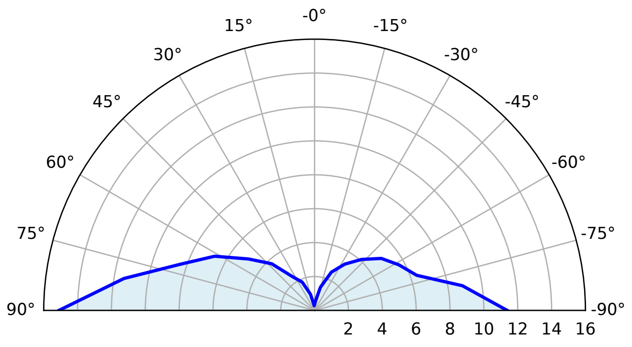


Figure 4.5: Polar plot of the desired yaw angle vs the achieved one

4.1.4 Feasible Joint Trajectories

The results discussed so far have reinforced the quality of the performance achieved by the learned policy in executing omnidirectional jumps. We have highlighted multiple times that the policy tends to be conservative when handling longer-distance jumps due to its safety-oriented behavior. To further evaluate the robustness and feasibility of the policy, Fig. 4.6, 4.7, and 4.8 present the desired and actual trajectories for joint positions, joint velocities, and joint torques, respectively, for all four legs during a forward jump of 0.6 meters. This distance serves as an excellent baseline, given the discussions so far.
As observed from the figures, there is no instance where any of the system's joint limits are violated, and the torque trajectory is tracked accurately. This ensures that even during high-demand maneuvers, the policy remains within the physical constraints of the system, demonstrating that no higher torque effort is required than what the system can generate. The precise tracking of both joint positions and velocities further validates the controller's effectiveness in maintaining system stability throughout the jump.

These results strongly support the argument that our method ensures physical feasibility. This has been achieved, in part, thanks to the careful injection of prior domain knowledge into both the action parameterization and reward engineering processes. By incorporating these elements into the learning framework, we can enforce realistic constraints, ensuring the robot's performance remains feasible under real-world conditions while optimizing for safety and robustness.

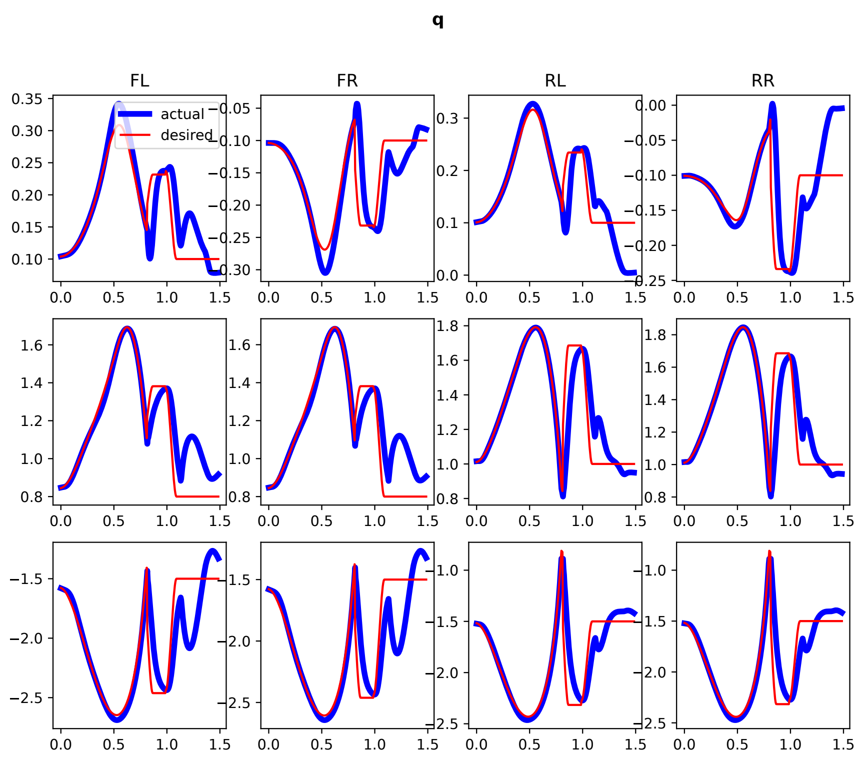


Figure 4.6: Actual and Desired joint position trajectories

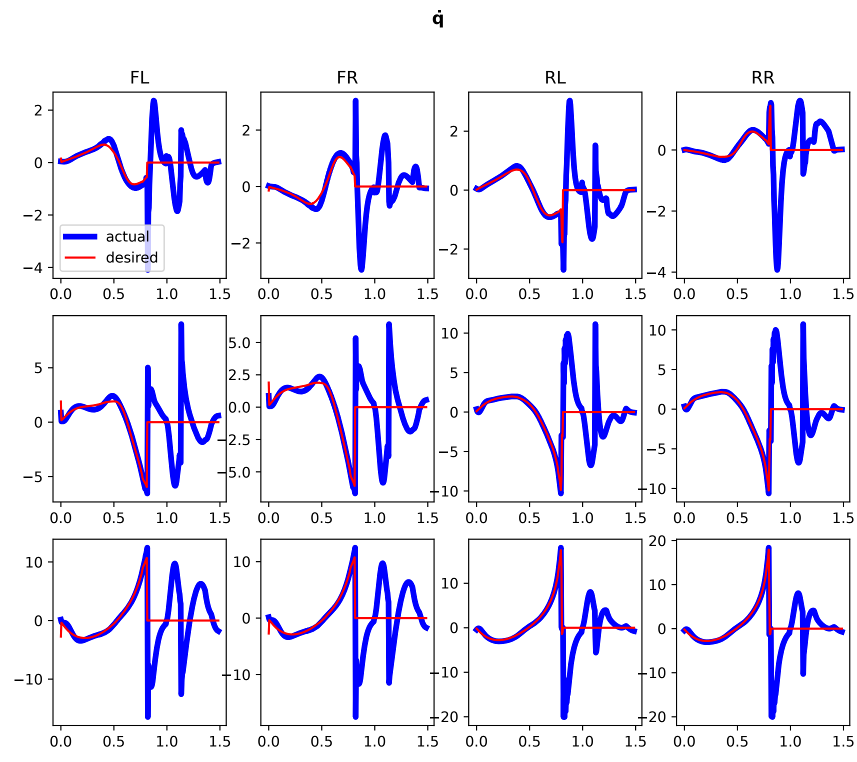


Figure 4.7: Actual and Desired joint velocity trajectories

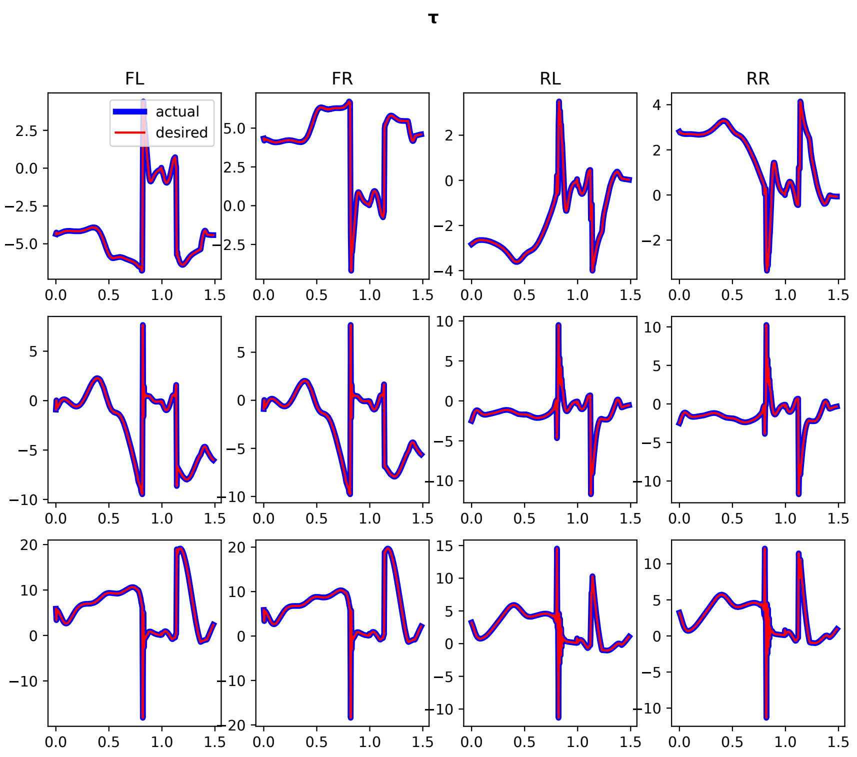


Figure 4.8: Actual and Desired torque position trajectories

4.1.5 Comparation with SoA Approach

To assess the quality of the proposed approach, we compare our method with the state-of-the-art E2E approach developed by Atanassov et al. [37], a collaborative effort between the Oxford Robotics Institute and the Department of Cognitive Robotics at Delft University of Technology. Briefly, their approach involves dividing the learning process into three curriculum stages, with the same policy, represented by a neural network, being refined sequentially. The first stage focuses on in-place jumps with variable positive height displacements. The second stage introduces forward and lateral displacement jumps, while the final stage involves obstacle-aware jumps, where the robot learns to jump onto a given obstacle.

The work was developed using an early version of Isaac-Gym in combination with a predecessor framework of Orbit, and the total training time took approximately 10 hours. Like our work, their training utilized parallelization with 4096 simulated robots running simultaneously. However, in their case, each policy update was computed after 24 environment steps, resulting in a higher data requirement compared to our method. The entire training process required 23,000 iterations, with 3,000 iterations for the first stage and 10,000 for each of the remaining two stages. This highlights the high data requirements of E2E approaches compared to our more sample-efficient GRL method. The training was performed on a 3090 GPU with 24 GB VRAM, providing a robust computational platform more powerful than the one utilized by us.

In Fig. 4.9, we present the results of their approach for forward jumps, while in Fig. 4.10, we show our results using a similar form factor for equal comparison. For consistency, we retrained our policy by utilizing their training region composed of an X-range of 0-1.2 meters and a Y-range of -0.3 to 0.3 meters.
A key observation is that, in their approach, distant jumps combined with lateral displacements often result in failed jumps due to improper contacts that are not made by the feet. In contrast, our approach avoids such failures by ensuring the respect of system constraints also in the task space. While this results in a slightly shorter maximum jump distance compared to their method, our approach

offers a more uniform performance within the previously defined feasible region, as evidenced by the proximity of our jump distributions to the ideal jumping behavior.
Although their plot does not explicitly report jump heights, in the paper they declare that their method was tested with jump heights of only 0.05 and 0.1 meters, whereas our method demonstrated the ability to reach up to 0.26 meters for forward jumps This highlights our method's superior capability for handling higher vertical displacements, despite being more conservative in terms of lateral displacement.
Overall, both methods exhibit strengths and weaknesses. The approach by Atanassov et al. enables longer jumps but at the cost of violating system constraints, such as maintaining proper foot contact. In contrast, our method adopts a more conservative strategy that prioritizes safety and accuracy, with the added advantage of requiring significantly fewer samples and avoiding complex multi-stage reward engineering. This makes our approach not only more data-efficient but also more straightforward to implement and train, without sacrificing real-world feasibility and performance. In addition to this characteristic, their approach incorporates 25 distinct reward terms tailored to each curriculum stage. In contrast, our approach uses only 15 reward terms, further demonstrating the simplicity and efficiency of our formulation.

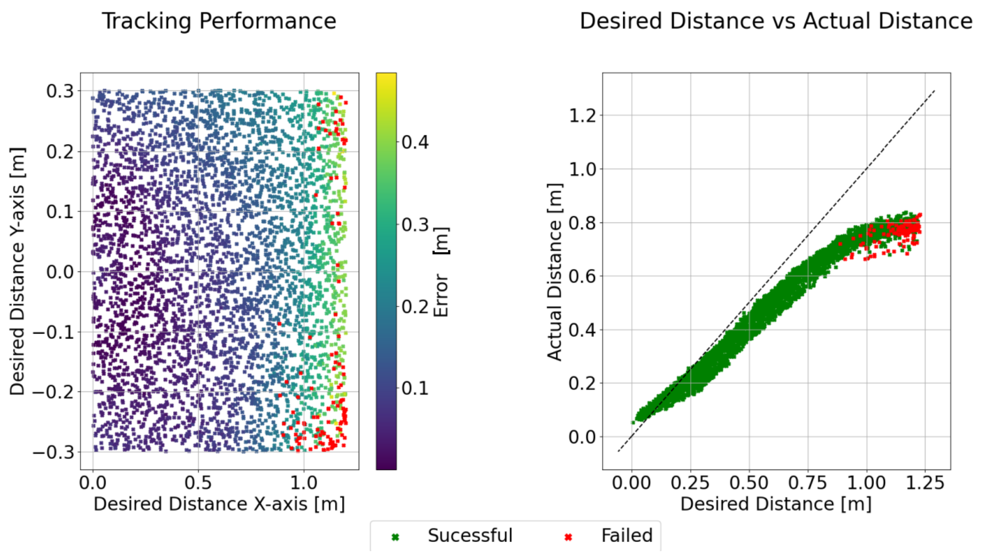


Figure 4.9: Jumping performances reported in Atassanov et al. work [37]

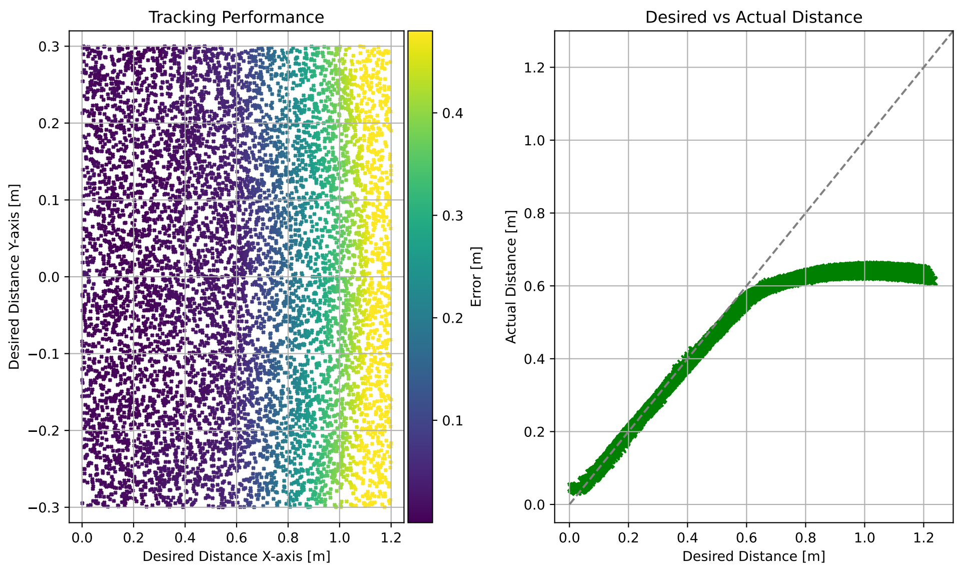
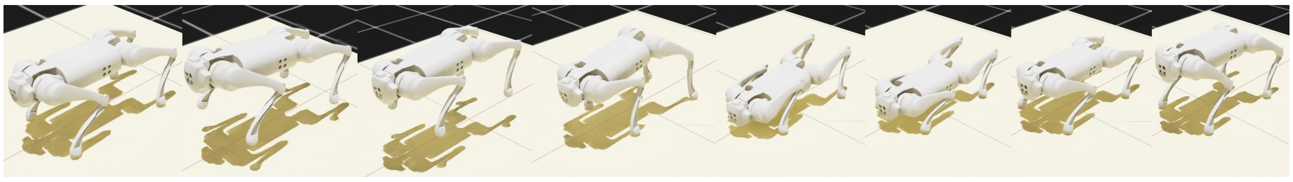


Figure 4.10: Jumping performances reported four our approach for comparison with state-of-the-art

4.1.6 Zero-Shot on Different RoboticPlatform

To demonstrate the generalization of our approach, we selected a quadruped platform similar to the Go1 and performed a zero-shot jump with this platform, making only the necessary low-level control adjustments. The selected platform is the Unitree Aliengo, which weighs approximately 21 kg and features longer leg links with higher torque limits compared to the Go1. As shown in Fig. 4.11, we tested a forward jump of 0.6 m, achieving a landing error of 0.1 m while satisfying all system constraints related to the robot's characteristics.
This underscores the potential for directly applying the trained policy to robotic systems with similar specifications, or alternatively, for rapidly fine-tuning the pre-trained policy to adapt to differences in actuation, mass, and dimensional characteristics across platforms. By defining both the action space and observation space in task-space, our approach contributes to making the policy largely independent of the specific robotic platform. Ultimately, this demonstrates the strength of our method in fostering platform-agnostic learning and control, supporting efficient transfer and broad applicability in real-world robotic systems.


(a) Jump executed on the Unitree Go1 robot

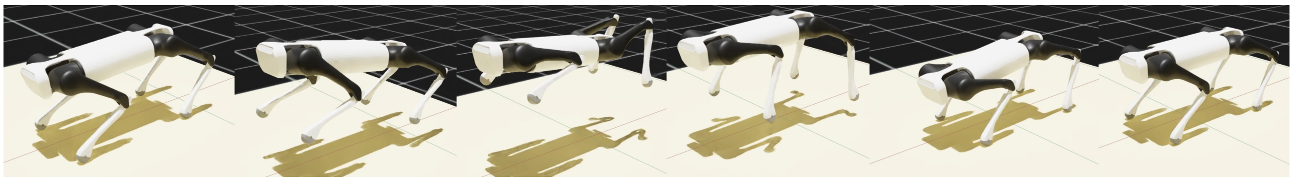

(b) Zero-shot jump executed on the Unitree Aliengo robot

Figure 4.11: Forward jump of 0.6 meters executed with the same policy on two robotic platforms

5 Conclusions and Future Works

In this thesis, we proposed a novel approach for enabling real-time omnidirectional jumping in quadruped robots using guided reinforcement learning (GRL). The core objective was to develop a policy that handles various jump configurations, including flat, vertical, and rotational jumps—while ensuring physical feasibility and safety. This was achieved by integrating system constraints directly into the learning process and leveraging domain-specific knowledge in both action space parametrization and reward function design. The result is a policy that achieves precise task execution while adhering to safe operational limits, striking a crucial balance on accuracy while respecting constraint satisfaction.

A significant contribution of this work is the two-stage thrust trajectory parametrization, which combines the adaptability of Bézier curves for the initial phase of the jump with a physics-based Uniformly Accelerated Rectilinear Motion (UARM) model for the final, more explosive phase. This approach not only enhances control over jump execution but ensures that each jump remains dynamically feasible within the robot's physical limits. By extending the action space to include both linear and angular components, the policy enables complex aerial maneuvers, such as twist jumps, and mid-air reorientation in preparation for landing.
Through extensive simulation testing, the policy demonstrated high accuracy and reliability for jumps up to 0.6 meters within a well-defined feasible region. Additionally, it performed effectively in vertical displacement scenarios, achieving jumps up to 0.26 meters upward and 0.4 meters downward. These capabilities are essential for real-world applications, especially when navigating uneven terrains. The policy also successfully handled orientation changes during jumps, demonstrating robust control over yaw angles with minimal error, which is critical for enhancing versatility in dynamic environments.
A key strength of our approach lies in its safety-oriented design. Unlike state-of-the-art end-to-end (E2E) methods [37], which can compromise system constraints to achieve long-distance jumps, our method ensures that all jumps respect the robot's physical limitations. This focus on feasibility and safety results in more reliable performance, even in complex scenarios. Furthermore, our method is significantly more sample-efficient, achieving comparable or superior performance with far fewer training episodes and samples.

To validate the generalization of the policy, we successfully performed a zero-shot transfer from the Unitree Go1 robot to the Unitree AlienGo robot, demonstrating that with minimal adjustments, the policy could be adapted to different quadruped platforms. The successful forward jump obtained, highlights the adaptability and robustness of our method, proving its potential for use across a range of robots with similar actuation and system characteristics.
In summary, the key benefits of our guided RL approach are its ability to prioritize safety through constraint-aware policy design, efficient thrust trajectory parametrization that ensures both precision and feasibility, and the significant sample efficiency gained by incorporating domain knowledge. The performance results underscore the value of these contributions, as the policy consistently maintained accurate, safe, and physically feasible jumps across various configurations and environments.

Looking ahead, this work establishes a strong foundation for future research in jumping quadruped locomotion, with several exciting potential extensions. One key improvement would be to remove the current limitation of keeping all four legs in contact with the ground during the thrust phase. By allowing for more dynamic movement, such as utilizing only two legs for rearing, we could unlock the ability for advanced maneuvers like backflips or barrel-rolls.
Another significant avenue for future research is the integration of a reactive landing controller directly

into the training phase. Currently, the control of the landing phase is simplified; adding a landing controller (potentially via a second neural network dedicated to learning in-air adjustments and landing stability) could dramatically improve performance during more dynamic landings. This would increase the robot's ability to recover from jumps with high velocities or in challenging environments. Along with this, adding the capability for the robot to jump from a non-zero initial velocity would allow for even more complex and agile maneuvers, particularly in environments where the robot is in continuous motion.

In terms of broader application, this work can be extended beyond quadrupeds to humanoid robots, allowing for more advanced robotic systems to benefit from the proposed methodology. The flexibility and adaptability demonstrated in the current work suggest that the same guided reinforcement learning approach could be generalized to other robotic forms, paving the way for humanoid robots to perform dynamic jumps and other complex locomotion tasks.

Finally, transitioning from simulation to real-world testing is a crucial next step to fully validate the practical applicability of the proposed policy. We plan to carry out real-world experiments in the near future, with the aim of publishing a paper based on this thesis. These experiments will test the robustness of the learned policy on the actual robot, highlighting potential areas for refinement, such as handling uneven terrains, adapting to unpredictable environments, and dealing with more complex physical interactions. This real-world validation will be pivotal for demonstrating the policy's effectiveness in practical applications such as exploration, search-and-rescue, and industrial automation, where agile and reliable robot mobility is essential.

By pursuing these avenues, future work can push the boundaries of robotic locomotion, creating more dynamic, flexible, and intelligent systems capable of navigating complex real-world environments.

Bibliography

[1] Chen Tang, Ben Abbatematteo, Jiaheng Hu, Rohan Chandra, Roberto Martín-Martín, and Peter Stone. Deep reinforcement learning for robotics: A survey of real-world successes. *arXiv preprint arXiv:2408.03539*, 2024.

[2] Priyaranjan Biswal and Prases K Mohanty. Development of quadruped walking robots: A review. *Ain Shams Engineering Journal*, 12(2):2017–2031, 2021.

[3] Fabian Jenelten, Ruben Grandia, Farbod Farshidian, and Marco Hutter. Tamols: Terrain-aware motion optimization for legged systems. *IEEE Transactions on Robotics*, 38(6):3395–3413, 2022.

[4] David Hoeller, Nikita Rudin, Dhionis Sako, and Marco Hutter. Anymal parkour: Learning agile navigation for quadrupedal robots. *Science Robotics*, 9(88):eadi7566, 2024.

[5] Francesco Roscia, Michele Focchi, Andrea Del Prete, Darwin G. Caldwell, and Claudio Semini. Reactive landing controller for quadruped robots. *IEEE Robotics and Automation Letters*, 8(11):7210–7217, 2023.

[6] Michele Focchi, Mohamed Bensaadallah, Marco Frego, Angelika Peer, Daniele Fontanelli, Andrea Del Prete, and Luigi Palopoli. Clio: a novel robotic solution for exploration and rescue missions in hostile mountain environments. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7742–7748, 2023.

[7] Hae-Won Park, Patrick M Wensing, and Sangbae Kim. High-speed bounding with the mit cheetah 2: Control design and experiments. *The International Journal of Robotics Research*, 36(2):167–192, 2017.

[8] Justin K. Yim, Bajwa Roodra Pratap Singh, Eric K. Wang, Roy Featherstone, and Ronald S. Fearing. Precision robotic leaping and landing using stance-phase balance. *IEEE Robotics and Automation Letters*, 5(2):3422–3429, 2020.

[9] Chuong Nguyen and Quan Nguyen. Contact-timing and trajectory optimization for 3d jumping on quadruped robots. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11994–11999, 2022.

[10] Jiatao Ding, Vassil Atanassov, Edoardo Panichi, Jens Kober, and Cosimo Della Santina. Robust quadrupedal jumping with impact-aware landing: Exploiting parallel elasticity. *IEEE Transactions on Robotics*, 2024.

[11] Benjamin Katz, Jared Di Carlo, and Sangbae Kim. Mini cheetah: A platform for pushing the limits of dynamic quadruped control. In *2019 international conference on robotics and automation (ICRA)*, pages 6295–6301. IEEE, 2019.

[12] Matthew Chignoli and Sangbae Kim. Online trajectory optimization for dynamic aerial motions of a quadruped robot. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7693–7699. IEEE, 2021.

[13] Gabriel García, Robert Griffin, and Jerry Pratt. Time-varying model predictive control for highly dynamic motions of quadrupedal robots. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7344–7349, 2021.

[14] Matthew Chignoli, Savva Morozov, and Sangbae Kim. Rapid and reliable quadruped motion planning with omnidirectional jumping. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 6621–6627, 2022.

[15] Carlos Mastalli, Wolfgang Merkt, Guiyang Xin, Jaehyun Shim, Michael Mistry, Ioannis Havoutis, and Sethu Vijayakumar. Agile maneuvers in legged robots:a predictive control approach. *ArXiv*, 2022.

[16] He Li and Patrick M. Wensing. Cafe-mpc: A cascaded-fidelity model predictive control framework with tuning-free whole-body control, 2024.

[17] Guillaume Bellegarda, Milad Shafiee, Merih Ekin Özberk, and Auke Ijspeert. Quadruped-frog: Rapid online optimization of continuous quadruped jumping. *arXiv preprint arXiv:2403.06954*, 2024.

[18] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[19] Christian Gehring, Stelian Coros, Marco Hutter, Carmine Dario Bellicoso, Huub Heijnen, Remo Diethelm, Michael Bloesch, Péter Fankhauser, Jemin Hwangbo, Mark Hoepflinger, et al. Practice makes perfect: An optimization-based approach to controlling agile motions for a quadruped robot. *IEEE Robotics & Automation Magazine*, 23(1):34–43, 2016.

[20] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[21] Xue Peng, Erwin Coumans, Tingnan Zhang, Tsang-Wei Lee, Jie Tan, and Sergey Levine. Learning agile robotic locomotion skills by imitating animals. *Robotics: Science and Systems*, 07 2020.

[22] Gwanghyeon Ji, Juhyeok Mun, Hyeongjun Kim, and Jemin Hwangbo. Concurrent training of a control policy and a state estimator for dynamic and robust legged locomotion. *IEEE Robotics and Automation Letters*, 7(2):4630–4637, 2022.

[23] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

[24] Peter Fankhauser, Marco Hutter, Christian Gehring, Michael Bloesch, Mark A Hoepflinger, and Roland Siegwart. Reinforcement learning of single legged locomotion. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 188–193. IEEE, 2013.

[25] Inc OpenAI. Benchmarks for spinning up implementations, 2022. `https://spinningup.openai.com/en/latest/spinningup/bench.html#benchmarks-for-spinning-up-implementations` [Accessed: 26/02/2023].

[26] Miroslav Bogdanovic, Majid Khadiv, and Ludovic Righetti. Model-free reinforcement learning for robust locomotion using demonstrations from trajectory optimization. *Frontiers in Robotics and AI*, 9:854212, 2022.

[27] Guillaume Bellegarda, Chuong Nguyen, and Quan Nguyen. Robust quadruped jumping via deep reinforcement learning, 2023.

[28] Gianluigi Grandesso, Elisa Alboni, Gastone P Rosati Papini, Patrick M Wensing, and Andrea Del Prete. Cacto: Continuous actor-critic with trajectory optimization—towards global optimality. *IEEE Robotics and Automation Letters*, 8(6):3318–3325, 2023.

[29] Xue Bin Peng and Michiel van de Panne. Learning locomotion skills using deeprl: Does the choice of action space matter? In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2017.

[30] Guillaume Bellegarda and Katie Byl. Training in task space to speed up and guide reinforcement learning. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2693–2699, 2019.

[31] Shuxiao Chen, Bike Zhang, Mark W Mueller, Akshara Rai, and Koushil Sreenath. Learning torque control for quadrupedal locomotion. In *2023 IEEE-RAS 22nd International Conference on Humanoid Robots (Humanoids)*, pages 1–8. IEEE, 2023.

[32] Michel Aractingi, Pierre-Alexandre Léziart, Thomas Flayols, Julien Perez, Tomi Silander, and Philippe Souères. Controlling the solo12 quadruped robot with deep reinforcement learning. *scientific Reports*, 13(1):11945, 2023.

[33] Milad Shafiee, Guillaume Bellegarda, and Auke Ijspeert. Manyquadrupeds: Learning a single locomotion policy for diverse quadruped robots. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3471–3477. IEEE, 2024.

[34] Amjad Yousef Majid, Serge Saaybi, Tomas van Rietbergen, Vincent François-Lavet, R. V. Prasad, and Chris Verhoeven. Deep reinforcement learning versus evolution strategies: A comparative survey. *ArXiv*, abs/2110.01411, 2021.

[35] Yuxiang Yang, Xiangyun Meng, Wenhao Yu, Tingnan Zhang, Jie Tan, and Byron Boots. Continuous versatile jumping using learned action residuals. In *Learning for Dynamics and Control Conference*, pages 770–782. PMLR, 2023.

[36] Francecso Vezzi, Jiatao Ding, Antonin Raffin, Jens Kober, and Cosimo Della Santina. Two-stage learning of highly dynamic motions with rigid and articulated soft quadrupeds. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9720–9726. IEEE, 2024.

[37] Vassil Atanassov, Jiatao Ding, Jens Kober, Ioannis Havoutis, and Cosimo Della Santina. Curriculum-based reinforcement learning for quadrupedal jumping: A reference-free design, 2024.

[38] Julian Eßer, Nicolas Bach, Christian Jestel, Oliver Urbann, and Sören Kerner. Guided reinforcement learning: A review and evaluation for efficient and effective real-world robotics [survey]. *IEEE Robotics & Automation Magazine*, 30(2):67–85, 2022.

[39] Riccardo Bussola, Michele Focchi, Andrea Del Prete, Daniele Fontanelli, and Luigi Palopoli. Efficient reinforcement learning for 3d jumping monopods. *Sensors*, 24(15):4981, 2024.

[40] Unitree go1. https://www.unitree.com/go1/. [Accessed: 19/09/2024].

[41] Unitree github repo. https://github.com/unitreerobotics. [Accessed: 19/09/2024].

[42] Justin Carpentier, Rohan Budhiraja, and Nicolas Mansard. Proximal and Sparse Resolution of Constrained Dynamic Equations. In *Robotics: Science and Systems 2021*, Austin / Virtual, United States, July 2021.

[43] Xuxin Cheng, Kexin Shi, Ananye Agarwal, and Deepak Pathak. Extreme parkour with legged robots. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11443–11450. IEEE, 2024.

[44] Clemens Schwarke, Victor Klemm, Matthijs Van der Boon, Marko Bjelonic, and Marco Hutter. Curiosity-driven learning of joint locomotion and manipulation tasks. In *Proceedings of The 7th Conference on Robot Learning*, volume 229, pages 2594–2610. PMLR, 2023.

[45] Dohyeong Kim, Hyeokjin Kwon, Junseok Kim, Gunmin Lee, and Songhwai Oh. Stage-wise reward shaping for acrobatic robots: A constrained multi-objective reinforcement learning approach. *arXiv preprint arXiv:2409.15755*, 2024.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[47] Inc OpenAI. Spinningup openai: Introduction to rl. `https://spinningup.openai.com/en/latest/spinningup/rl_intro.html` [Accessed: 26/02/2023].

[48] John Schulman. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

[49] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[50] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

[51] Isaac-sim. https://developer.nvidia.com/isaac/sim. [Accessed: 19/09/2024].

[52] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, et al. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023.

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

[54] RSL Lab. Rsl rl, 2021. `https://github.com/leggedrobotics/rsl_rl` [Accessed: 18/09/2024].

[55] Gabriel B Margolis and Pulkit Agrawal. Walk these ways: Tuning robot control for generalization with multiplicity of behavior. *Conference on Robot Learning*, 2022.

[56] Michele Focchi, Francesco Roscia, and Claudio Semini. Locosim: an open-source cross-platform robotics framework. In *Climbing and Walking Robots Conference*, pages 395–406. Springer, 2023.

[57] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[58] Felix Grimminger, Avadesh Meduri, Majid Khadiv, Julian Viereck, Manuel Wüthrich, Maximilien Naveau, Vincent Berenz, Steve Heim, Felix Widmaier, Thomas Flayols, et al. An open torque-controlled modular robot architecture for legged locomotion research. *IEEE Robotics and Automation Letters*, 5(2):3650–3657, 2020.

[59] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[60] Rohan Budhiraja, Justin Carpentier, Carlos Mastalli, and Nicolas Mansard. Differential dynamic programming for multi-phase rigid contact dynamics. In *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, pages 1–9. IEEE, 2018.

[61] Yasuhiro Fujita and Shin-ichi Maeda. Clipped action policy gradient. In *International Conference on Machine Learning*, pages 1597–1606. PMLR, 2018.