Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Execution of agile motions with a lightweight quadruped robot via Optimal Control

Relatore
Chiar.mo Prof. Bruno Siciliano

Correlatori
Prof. Joan Solà
Josep Martí-Saumell, PhD
Hugo Duarte

Candidato
Giuliano de Julio
P38/84

Anno Accademico 2022–2023

Contents

1. Introduction

1.1 Thesis statement and contents organization

The thesis is the result of a six-month internship at the HiDRo research group of the Institut de Robòtica i Informàtica Industrial (IRI), Barcelona. The research group acquired the lightweight open-source quadruped robot Solo-12 from PAL Robotics in 2022 for research in Optimal Control and State Estimation.

The goal of the thesis is to investigate ways to generate agile motions based on Optimal Control techniques. These methods were first tested in simulation and then transferred to the real robot. Relevant motion skills that have been be pursued are:

- Moving the robot base in 6 DoF with 4 feet static on ground;
- Jumping with 4 feet at once;
- Backflip (only in simulation)

The thesis is divided into four chapters: Chapter 1 presents the robot hardware and some modifications made in the laboratory to extend its capabilities. In Chapter 2 we discuss the dynamic model and how it is used to determine the static joint torques to make the robot stand in a default configuration. The low-level impedance controller is also introduced. In Chapter 3 we define the Optimal Control problem and how it is solved using the Differential Dynamic Programming algorithm to generate agile motions such as the jump and the backflip. In Chapter 4, we present the ROS2 application deployed on the real robot, as well as some experiments performed to validate the theoretical results. Finally, we present conclusions and future perspectives.

1.2 Performing a backflip: Trajectory optimization vs Reinforcement Learning

While Mini Cheetah by Massachusetts Institute of Technology defined the first four-legged robot in the world to do a backflip by using offline trajectory optimization [KCK19], Solo-8 by Open Dynamic Robot Initiative caught up by achieving this agile maneuver with Reinforcement Learning [Li+22].

Mini Cheetah's backflip is the solution to a huge nonlinear offline trajectory optimization problem. This model-based method specifies a trajectory in which the robot would start in a given right-side-up orientation and end up flipped 360 degrees along the pitch axis. The algorithm solves all the torques to be applied by each motor at each joint, at each time step between start and end, to perform the backflip.

Solo's backflip is the result of solving an imitation objective using a model-free reinforcement learning method. The robot imitates base motion reference trajectories from a human demonstrator, who manually holds and flips the robot in his hands. The imitation learning algorithm trains thousands of parallel learning agents in simulation and returns high rewards when they approach the backflip trajectories demonstrated by the human operator. Since reference trajectories are given only for the floating base motions, Solo's legs movements are completely figured out by the algorithm.

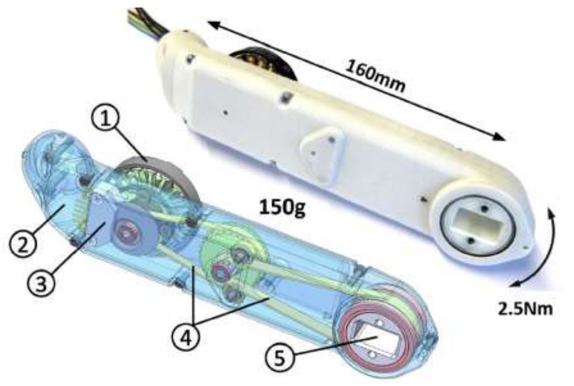
Both methods have been shown to be effective, however they can fail drastically:

- Trajectory optimization relies on accurate models of the robot's dynamics. It may fail due to poor model quality and also because of the non-linear dynamics. In such cases, the optimization solver may not be able to find and optimal solution that accurately models the robot's behavior.
- Reinforcement learning and sim2real deployment of the learned controller rely on the accuracy of the physics in the simulator. It may fail when the learned controller is unable to deal with the inaccuracy and uncertainty in the rigid body dynamics and environment.

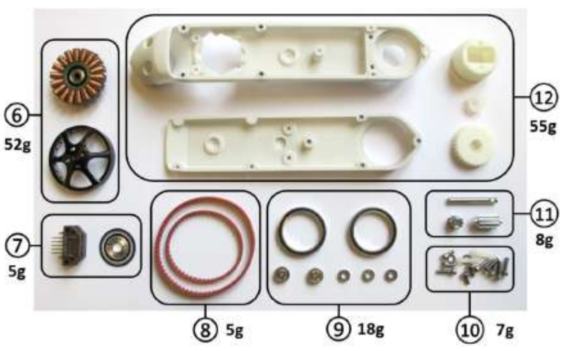
1.3 The Open Dynamic Robot Initiative

The Open Dynamic Robot Initiative (ODRI) project started in 2019 as a collaboration between the Motion Generation and Control Group, the Dynamic Locomotion Group and the Robotics Central Scientific Facility at the Max Planck Institute for Intelligent System, the Machines in Motion Laboratory at New York University's Tandon School of Engineering and the Gepetto Team at the Laboratory of Architecture and Analysis of Systems LAAS-CNRS. This project originated in an effort to build a low-cost, low-complexity actuator module using brushless motors that can be used to build various types of torque-controlled robots using mostly 3D-printed and off-the-shelf components. This module and extensions can be used to build legged robots or manipulators, with the intention of providing the community with reliable platforms that are easy to maintain and repair, and that could benefit from many contributions in their development. All hardware and software has been open sourced under the BSD 3-clause license so that the robots can be easily reproduced by other research laboratories. The entry point of the resources made available is [ODR].

The actuator module consists of a brushless motor connected to a 9:1 dual-stage timing belt transmission to provide a reasonable peak torque and high velocity without going over-



(a) Assembled actuator module



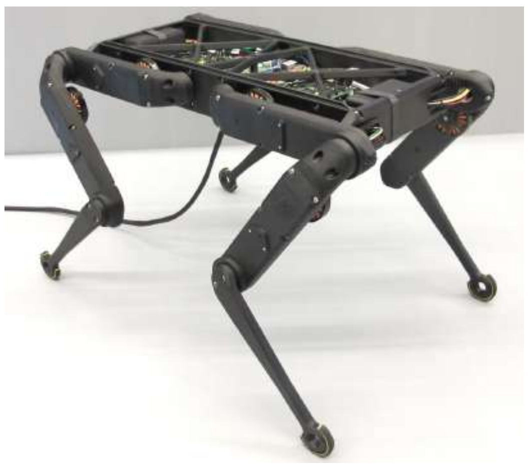
(b) Component overview

Figure 1.1: Brushless actuator module (a) assembled, and (b) individual parts. Brushless DC motor ①, two-part 3D printed shell structure ②, high resolution encoder ③, timing belts ④, and output shaft ⑤. Brushless motor ⑥, optical encoder ⑦, timing belts ⑧, bearings ⑨, fasteners ⑩, machined parts ⑪ and 3D printed parts ⑫. Figures extracted from [Gri+20].

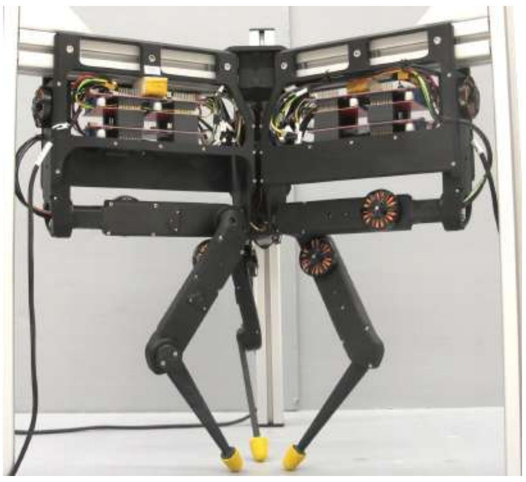
the-top. The actuator is capable of delivering 2.7 Nm joint torque at 12 A. A high-resolution optical encoder and a 5000 count-per-revolution coding wheel mounted on the motor shaft provide joint position measurements. Unlike absolute encoders, this incremental encoder requires a short start-up procedure to locate the index of the wheel. The whole actuator weights 150 g for a segment length of 16 cm. Except for the motor shaft and timing belts, all parts are either 3D printed or can be purchased off the shelf. An assembled view of the actuators and of its individual parts is shown in Fig. 1.1.

The drivers can run an onboard impedance controller at 10 kHz and operate at motor voltages up to 40 V. The driver boards are managed by a single master board that handles communication with the control computer, either wired or wireless.

The first two platforms built under ODRI were the Solo-8, a lightweight (1.9 kg) quadruped with 8 degrees of freedom (2 per leg), and the TriFinger, a manipulator platform with 3 identical finger modules, each with 3 actuators. Both platforms are shown in Fig. 1.2.



(a) Solo-8 quadruped



(b) TriFinger Manipulator Platform

Figure 1.2: Both platforms highlight the possibilities offered even with standardized actuation modules, either for legged robotics or for object manipulation. Figures extracted from [ODR].

1.4 The Solo-12 quadruped

Further development in 2020 resulted in an enhanced version of the Solo-8 quadruped, called the Solo-12 due to its 12 actuators (3 per leg). Otherwise, it is structurally very similar to its predecessor, as seen in Fig. 1.3.

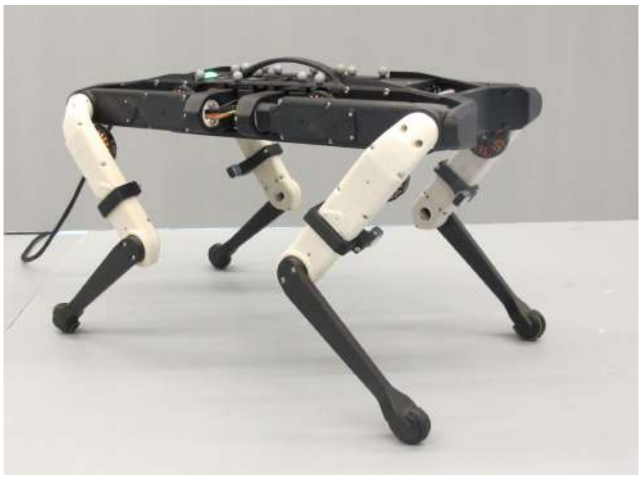


Figure 1.3: Solo-12 in its default configuration

Compared to Solo-8 which had only 2 actuators per leg for the Hip Flexion/Extension and the knee and thus could only move its feet in a vertical plane, Solo-12 eliminates this limitation thanks to Hip Abduction/Adduction. This greatly expands the workspace of the feet and thus the robot's capabilities, as it can now move laterally and stabilize itself better. It also opens up more possibilities for controlling the base orientation. The internal electronics remain similar to Solo-8, with only two additional micro-drivers to handle the four additional actuators. The workspace of each leg is highlighted in Fig. 1.4.

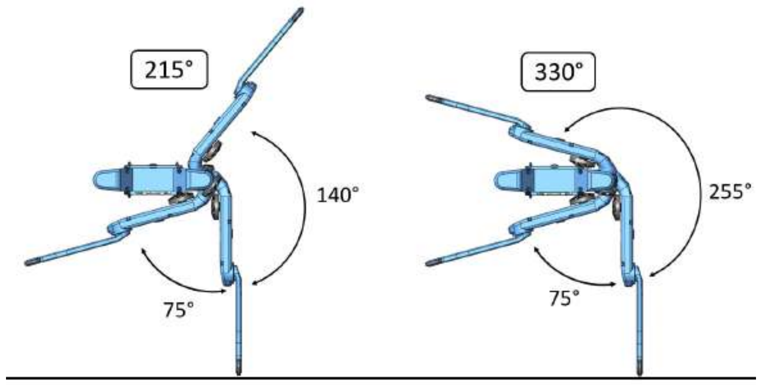


Figure 1.4: The standard range of motion of the Hip Abduction/Adduction is 215 deg. It extends up to 325 deg if the Hip Flexion/Extension is moved as well. Figure extracted from [ODR].

1.5 Extending Solo-12 capabilities

Essential capabilities for mobile robots are autonomy, exteroceptive sensing and high performance onboard computation. To meet these requirements, changes were made to the initial hardware configuration of IRI's Solo-12.

An overview of the initial hardware configuration of is shown in Fig. 1.5. With the exception of the body chassis, legs, and drive belts, the remaining hardware components are briefly described in the following table:

Component	Quantity	Basic description
Motors	12	The 12 leg joints are actuated by brushless DC (BLDC) motors capable of delivering up to 2.7 Nm of torque at 12 A..
Sensors	13	Proprioceptive sensing is provided by an inertial measurement unit (IMU) and incremental encoders.
Microdrivers	6	Each microdriver installed in Solo-12 drives two brushless motors and performs a torque control at 10kHz for each of them. Communication between the microdrivers and with the master board is via SPI at 1 kHz.
Master board	1	In a system consisting of several microdrivers, sensors, and a computer to communicate with, a master board is used to centralize the data provided by these elements. In our case, an Ethernet cable connected the master board to the computer.
Power distribution board	1	Any system with a main power source required by different hardware components should include a power distribution (PD) board, typically implemented on a printed circuit board (PCB). In addition, DC converters are required if the different hardware systems operate at different voltages. For example, all of the modules on the Solo-12 can operate between 22.2 and 24.0V. The PCB design is done by the HiDRo team and the DC/DC is already configured and ready to go.
Power supply	1	A power supply is required to power the electronic components and actuators. It can vary depending on the size, mobility and power requirements of the robot. In our case, we used the B&K Precision 1901B DC power supply, which can deliver a voltage of 24 V and a current of up to 30 A.
Computer	1	The computer enables complex calculations, advanced algorithms, and sophisticated data processing in real time. It can also handle computationally intensive tasks such as image processing, machine learning, or 3D mapping, which may be beyond the capabilities of a microcontroller alone. In our case, we used an MSI GS66 Stealth laptop running Ubuntu 20.04 LTS.

Table 1.1: Initial hardware configuration

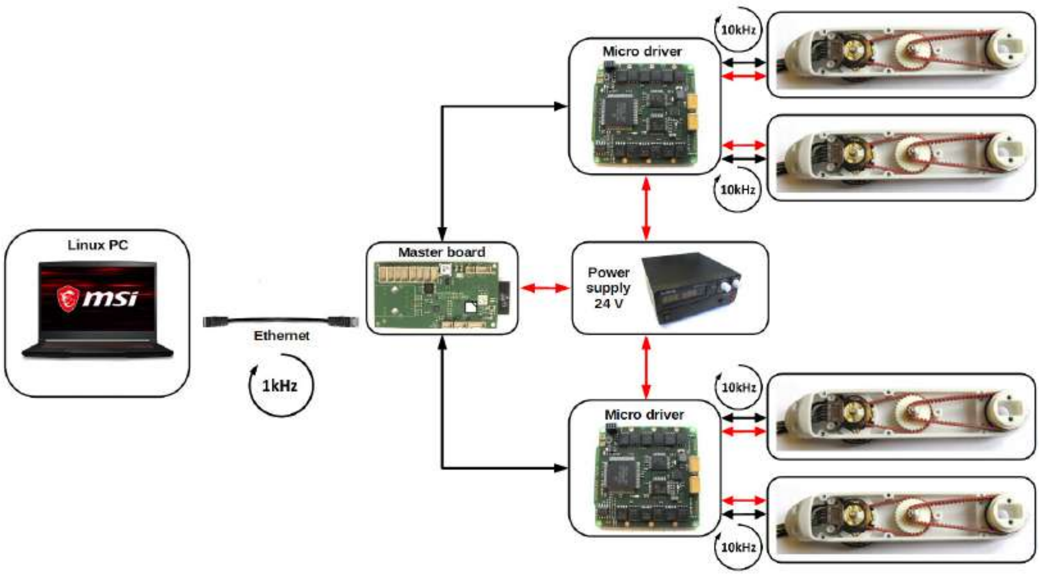


Figure 1.5: Initial hardware configuration overview. An ethernet cable connects a control computer to the master board at 1 kHz. The master board manages all microdrivers that each control 2 actuators at 10 kHz.

Batteries

Lithium Polymer (LiPo) are among the most widely used batteries for lightweight robots. This, and the fact that the group already had experience with them, was the main reason for their choice. Onboard batteries must provide the same voltage as the power supply described in Table 1.1. For a single 22.2V battery, most off-the-shelf batteries are either too large or too heavy for a lightweight robot. This leaves no choice but to use two or more smaller batteries. The final choice was a pair of LiPo TATTU R-Line 3s 11.1V 750 mAh 95C batteries, one to be placed on the front side and the other on the rear side of Solo-12.

Onboard computer

The selection of the on-board computer requires an in-depth knowledge of the specific electronics, which is beyond the scope of this work. However, some aspects can be highlighted that justify the choice of a candidate on-board computer. Firstly, the mass of the computer must be small: this leads to the so-called mini PCs. Mini PCs offer the same features as any laptop or desktop computer, with the advantage of being smaller. Like any other computer, when choosing one, it is necessary to know the desired features, such as the processor's computing power, how much RAM is needed, and which and how many ports will be used. Among the listed ones, the focus here needs to be on the processor. The main reason is the need to meet the computational demands of complex control algorithms such as Model Predictive Control (MPC). The chosen on-board computer is the Intel NUCi7DNKE with an Intel i7-8650U with 4 cores and a clock frequency of 1.9 GHz and 32 GB DDR4 RAM.

Sensors

State estimation, mapping and localization are some of the biggest challenges in mobile robotics. On the one hand, encoders and the IMU provide proprioceptive sensory data, and on the other hand, motion capture systems such as OptiTrack provide exteroceptive sensory data. Through sensor fusion of data from different sources, state estimation techniques such as the Extended Kalman Filter (EKF) can be implemented to estimate the state of the robot. Extending the capabilities of the robot is beyond the scope of this work, but in the future it is planned to add an image+depth camera (Intel RealSense Depth Camera D435i) and a LiDAR to the robot.

Table 1.2 summarizes the new hardware components and Fig. 1.6 shows the new hardware configuration overview. Figure 1.7 shows a comparison between the old and new CAD models.

Component	Quantity	Model name
Battery	2	LiPo TATTU R-Line 3s 11.1V 750 mAh 95C
Onboard computer	1	Intel NUCi7DNKE
Camera	1	Intel RealSense Depth Camera D435i

Table 1.2: New hardware components

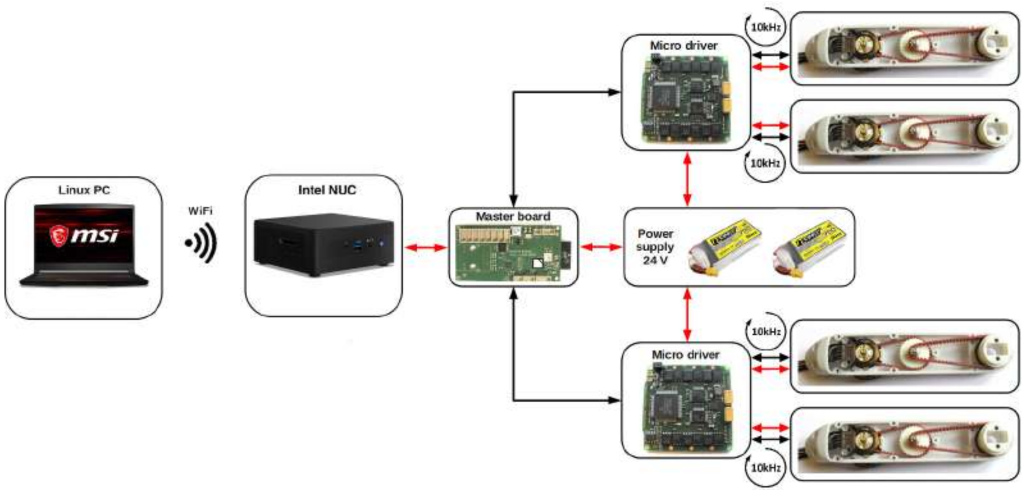
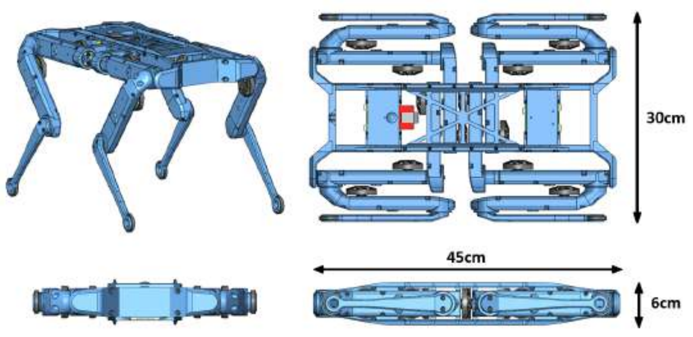
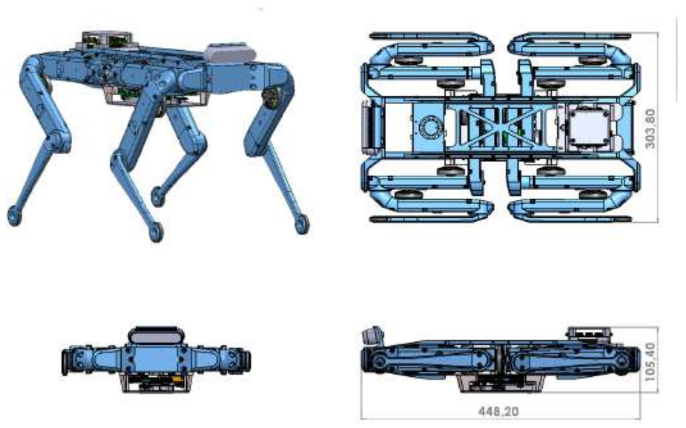


Figure 1.6: New hardware configuration overview. A WiFi connection using SSH protocol connects a laptop to access the onboard computer shell. The two batteries substitute the power supply described in Table 1.1.



(a) Old CAD model



(b) New CAD model

Figure 1.7: Comparison between the old and new CAD models.

2. Dynamic Model

As we can see in Fig. 2.1, the quadruped has a total of $n_v = 18$ DoFs: 6 unactuated DoFs for the floating base and $n_j = 12$ actuated DoFs for the joints. Like any mechanical system, the quadruped can be described by the state vector

$$\mathbf{x} = \begin{bmatrix} {}_{\mathcal{I}}\mathbf{q} \\ {}_{\mathcal{B}}\dot{\mathbf{q}} \end{bmatrix}, \tag{2.1}$$

which is $(2n_v + 1)$ - dimensional if the base orientation is described by quaternions, or $2n_v$ - dimensional if Euler angles are used. The configuration vector $\mathbf{q}$ contains the base position $\mathbf{q}_{b_P}$ and orientation $\mathbf{q}_{b_R}$ expressed in an inertial "world" frame $\mathcal{I}$, and joint angles $\mathbf{q}_j$. The velocity vector $\dot{\mathbf{q}}$ includes the base linear velocity $\dot{\mathbf{q}}_{b_P}$ and the angular velocity $\dot{\mathbf{q}}_{b_R}$ expressed in a local body frame $\mathcal{B}$, and the joint velocities $\dot{\mathbf{q}}_j$. Due to the different reference frames, $\dot{\mathbf{q}}$ is not a pure time derivative of $\mathbf{q}$, but requires an additional coordinate transformation $\mathbf{T}_{\mathcal{I}\mathcal{B}}$, which is consists of a pure rotation matrix for linear velocities as well as a slightly more complex mapping matrix for angular velocities.

$$\mathbf{q} = \begin{bmatrix} \mathbf{q}_{b_P} \\ \mathbf{q}_{b_R} \\ \mathbf{q}_j \end{bmatrix} \qquad \text{Configuration vector}$$

$$\mathbf{q}_{b_P} = \begin{bmatrix} x & y & z \end{bmatrix}^\top \qquad \text{Base position}$$

$$\mathbf{q}_{b_R} = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix}^\top \qquad \text{Base orientation (quaternions)}$$

$$\mathbf{q}_j = \begin{bmatrix} \theta_{\ell f,haa} & \theta_{\ell f,hfe} & \theta_{\ell f,kfe} & \cdots & \theta_{rh,kfe} \end{bmatrix}^\top \qquad \text{Joint angles}$$

$$\dot{\mathbf{q}} = \begin{bmatrix} {}_{\mathcal{I}}\mathbf{v}_B \\ {}_{\mathcal{B}}\boldsymbol{\omega}_{IB} \\ \dot{\mathbf{q}}_j \end{bmatrix} \qquad \text{Velocity vector}$$

The first part of the subscripts refers to the leg: ℓf is left-front, rf is right-front, ℓh is left-hind and rh is right-hind. The second part refers to the joint movements: haa for Hip Abduction/Adduction, hfe for Hip Flexion/Extension and kfe for Knee Flexion/Extension.

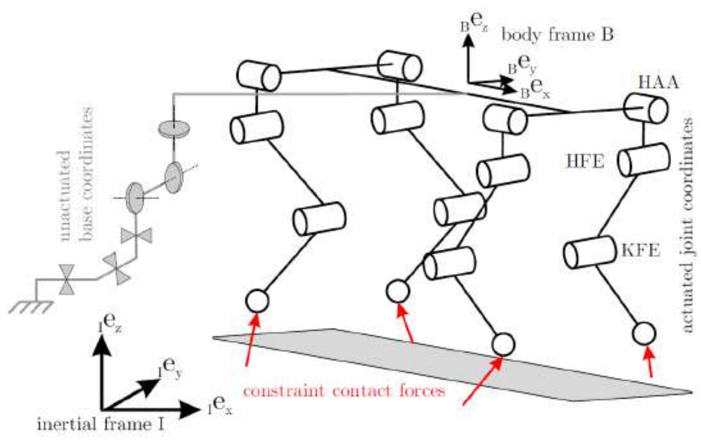


Figure 2.1: Quadruped structure

The equation of motion of the quadruped has the form

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) = \mathbf{S}^\top \boldsymbol{\tau} + \mathbf{J}_c^\top \boldsymbol{\lambda} \tag{2.2}$$

consisting of the following terms:

- $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{n_v \times n_v}$ is the inertia matrix;
- $\mathbf{c}(\mathbf{q}, \dot{\mathbf{q}})$ and $\mathbf{g}(\mathbf{q})$ are the Coriolis, centripetal and gravity terms;
- $\mathbf{S} \in \mathbb{R}^{n_j \times n_v}$ is a matrix selecting the actuated DoFs;
- $\boldsymbol{\tau} = \begin{bmatrix} \tau_{\ell f,haa} & \cdots & \tau_{rh,kfe} \end{bmatrix}^\top \in \mathbb{R}^{n_j}$ are the torques of the actuated DoFs;
- $\boldsymbol{\lambda} = \begin{bmatrix} \boldsymbol{\lambda}_{\ell f}^\top & \boldsymbol{\lambda}_{rf}^\top & \boldsymbol{\lambda}_{\ell h}^\top & \boldsymbol{\lambda}_{rh}^\top \end{bmatrix}^\top \in \mathbb{R}^{3n_c}$ collects the forces at the contact points exerted by the environment on the n_c stance feet;
- $\mathbf{J}_c^\top \in \mathbb{R}^{n_v \times 3n_c}$ is the contact Jacobian transpose mapping the forces at the contact points into joints torques:

$$\begin{aligned}\mathbf{J}_c^\top = \left[\begin{array}{c} \mathbf{J}_{c,b}^\top \\ \hdashline \mathbf{J}_{c,j}^\top \end{array}\right] &= \left[\begin{array}{c:c:c:c} \mathbf{J}_{c,\ell f,b}^\top & \mathbf{J}_{c,rf,b}^\top & \mathbf{J}_{c,\ell h,b}^\top & \mathbf{J}_{c,rh,b}^\top \\ \hdashline \mathbf{J}_{c,\ell f,j}^\top & \mathbf{J}_{c,rf,j}^\top & \mathbf{J}_{c,\ell h,j}^\top & \mathbf{J}_{c,rh,j}^\top \end{array}\right] \\ &= \left[\begin{array}{c:c:c:c} \mathbf{J}_{c,\ell f}^\top & \mathbf{J}_{c,rf}^\top & \mathbf{J}_{c,\ell h}^\top & \mathbf{J}_{c,rh}^\top \end{array}\right]\end{aligned}$$

The forward and inverse dynamics equations are:

$$\mathbf{S}^\top \boldsymbol{\tau} = \mathbf{ID}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) = \mathbf{M}\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) - \mathbf{J}_c^\top \boldsymbol{\lambda} \tag{2.3}$$

$$\ddot{\mathbf{q}} = \mathbf{FD}(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau}) = \mathbf{M}^{-1}(\mathbf{S}^\top \boldsymbol{\tau} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) + \mathbf{J}_c^\top \boldsymbol{\lambda}) \tag{2.4}$$

Finally, the overall system dynamics is

$$\dot{\mathbf{x}} = \begin{bmatrix} {}_{\mathcal{I}}\dot{\mathbf{q}} \\ {}_{\mathcal{B}}\ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \mathbf{T}_{\mathcal{IB}}\, {}_{\mathcal{B}}\dot{\mathbf{q}} \\ \mathbf{FD}(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau}) \end{bmatrix} \tag{2.5}$$

2.1 Computation of the static joints torques

The inverse dynamics equation can be used to calculate the nominal torque that the motors must apply in order to make the robot stand in a configuration with $n_c = 4$ feet on the ground, for example, the default configuration in Fig. 2.2. Since $\dot{\mathbf{q}}_0 = 0$ and $\ddot{\mathbf{q}}_0 = 0$ from (2.3) we have $\mathbf{S}^\top \boldsymbol{\tau} = \mathbf{g}(\mathbf{q}_0) - \mathbf{J}_c^\top \boldsymbol{\lambda}$, i.e.

$$\begin{bmatrix} \mathbf{0}_6 \\ \boldsymbol{\tau} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_b(\mathbf{q}_0) \\ \mathbf{g}_j(\mathbf{q}_0) \end{bmatrix} - \begin{bmatrix} \mathbf{J}_{c,b}^\top \\ \mathbf{J}_{c,j}^\top \end{bmatrix} \boldsymbol{\lambda} \tag{2.6}$$

From the first equation we can compute $\boldsymbol{\lambda} = (\mathbf{J}_{c,b}^\top)^\dagger \mathbf{g}_b(\mathbf{q}_0)$, and plug it into the second equation so to have:

$$\boldsymbol{\tau} = \mathbf{g}_j(\mathbf{q}_0) - \mathbf{J}_{c,j}^\top (\mathbf{J}_{c,b}^\top)^\dagger \mathbf{g}_b(\mathbf{q}_0) \tag{2.7}$$

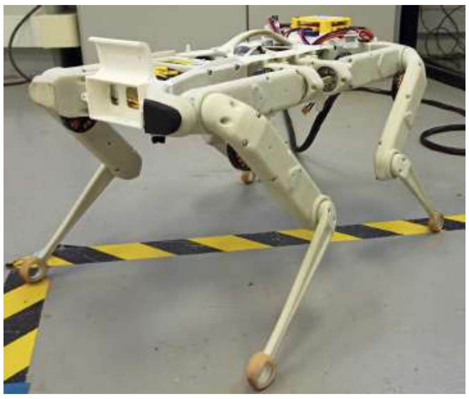


Figure 2.2: Default Configuration

2.2 Low-level impedance controller

The impedance controller is the low-level interface between the control architecture and the motor boards that drive the joints with a high-frequency current loop. It has a PD+ structure that takes reference torques, joint positions, and velocities as inputs and returns the torques to the actuators:

$$\boldsymbol{\tau}_{\text{mot}} = k_{ff} \boldsymbol{\tau}^\star + k_P(\mathbf{q}_j^\star - \mathbf{q}_j) + k_D(\dot{\mathbf{q}}_j^\star - \dot{\mathbf{q}}_j) \tag{2.8}$$

where $k_{ff} \in [0, 1]$ multiplies the feedforward term $\boldsymbol{\tau}^\star$, which is computed with (2.7) when the robot is still, and it is time-varying when one wants to track a given motion trajectory. In the latter case $\boldsymbol{\tau}^\star$, $\mathbf{q}_j^\star$ and $\dot{\mathbf{q}}_j^\star$ are the solution of an optimal control problem, as described in Chap. 3. Given $\boldsymbol{\tau}_{\text{load}} = R\boldsymbol{\tau}_{\text{mot}}$, being $R = 9$ the torque reduction ratio, these torques are then converted into current references using the motor torque constant $k_\tau = 0.025$ Nm/A:

$$\boldsymbol{i} = \frac{\boldsymbol{\tau}_{\text{mot}}}{k_\tau R} \tag{2.9}$$

where $\boldsymbol{i}$ collects all the quadrature currents that produce motor torques, each motor being controlled with Field Oriented Control. Using only target positions and velocities in (2.8) would force to have high k_P and k_D gains to move properly, since the torque commands would be generated only by tracking error. This would be similar to position control, with a very stiff behavior of the joints. On the other hand, torque control with $\boldsymbol{\tau}^{\star}$ alone would quickly diverge since these torques are calculated with a model that does not perfectly match the reality, so the robot would not move as expected due to torque constants that are not exactly the same for all motors, the elasticy of the mechanical structure, unmodeled phenomena, and so on.

Finally, combining torque commands with target positions and velocities is beneficial for two reasons. Compared to a pure torque command, taking into account position and velocity errors helps to correct for model inaccuracies and unexpected events. It also allows the use of lower gains because $\boldsymbol{\tau}^{\star}$ provides most of the torque amount even without any tracking error. As a result, we can achieve a more compliant behavior of the joints to dampen impacts with the ground.

3. Optimal Control Problem

An Optimal Control Problem (OCP) is an optimization problem involving a dynamical system and a cost function whose value depends on the evolution of the dynamical system. In its discrete-time version it has the form:

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & \sum_{k=0}^{N-1} \ell_k(\mathbf{x}_k, \mathbf{u}_k) + \ell_N(\mathbf{x}_N) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k), \quad k \in [0, N-1] \\ & \mathbf{x}_0 = \mathbf{x}(0) \end{aligned} \tag{3.1}$$

where $k+1 = k + \Delta t$, $N \in \mathbb{N}$ is the number of nodes, $\mathbf{X} = \{\mathbf{x}_0, \ldots, \mathbf{x}_N\}$ and $\mathbf{U} = \{\mathbf{u}_0, \ldots, \mathbf{u}_{N-1}\}$ are the state and control trajectories, $\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)$ is the numerical integration of (2.5), $\ell_k(\mathbf{x}_k, \mathbf{u}_k)$ is the running cost and $\ell_N(\mathbf{x}_N)$ is the terminal cost. The state $\mathbf{x} \in \mathcal{X}$ lies in a differential manifold (with dimension n_x) and $\mathbf{u} \in \mathbb{R}^{n_u}$ defines the input commands.

3.1 Cost function

The cost function describes the desired behavior of the system and it is an intuitive way to specify control objectives. It is composed of different *residuals*, i.e. costs on both state and control variables. We structure missions by a concatenation of *phases*, which we divide into two main types (Fig. 3.1):

- Task phases are the periods of time when the robot is required to do something specific or to be somewhere. Waypoints, contacts or manipulations are considered tasks. They are typically implemented by quadratic costs on (parts of) the state, or on functions of the state;
- Navigation phases correspond to the periods of time when the robot is free to move from one completed task to the next. They are mainly constrained by the robot dynamics model.

The duration T_h of the phase h is a design choice, and one should choose a duration that is consistent with feasible dynamics and control constraints. In general, the costs are written as the square-weighted 2-norms of various residuals, yielding least-squares formulations that

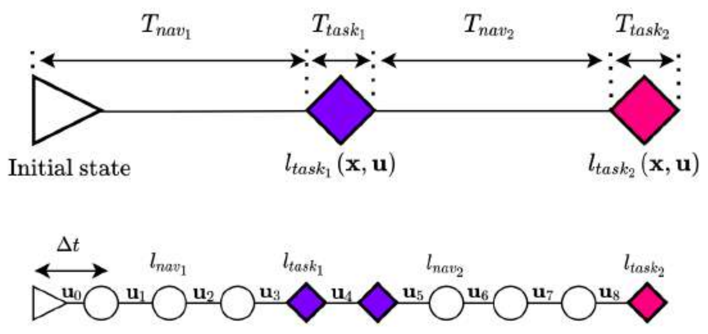

T_{nav_1}
T_{task_1}
T_{nav_2}
T_{task_2}
Initial state
$l_{task_1}(\mathbf{x},\mathbf{u})$
$l_{task_2}(\mathbf{x},\mathbf{u})$
Δt
l_{nav_1}
l_{task_1}
l_{nav_2}
l_{task_2}
$\mathbf{u}_0$ $\mathbf{u}_1$ $\mathbf{u}_2$ $\mathbf{u}_3$ $\mathbf{u}_4$ $\mathbf{u}_5$ $\mathbf{u}_6$ $\mathbf{u}_7$ $\mathbf{u}_8$

Figure 3.1: Top: trajectory specification considering two tasks and two navigation phases. Bottom: a possible translation to an OCP, containing a series of nodes uniformly distributed in time

are easy to handle. Thus, the cost function for the k-th node associated with the phase h with R_h associated residuals is

$$\ell_{k,h}(\mathbf{x}_k, \mathbf{u}_k) = \sum_{i=1}^{R_h} w_i ||\mathbf{r}_i(\mathbf{x}_k, \mathbf{u}_k)||^2_{\mathbf{W}_i} \tag{3.2}$$

where $\mathbf{r}_i(\mathbf{x}_k, \mathbf{u}_k)$ is a vector-valued residual specifying the i-th task for the k-th node, $\mathbf{W}_i$ is a square weighting matrix, usually diagonal, $||\mathbf{v}||^2_{\mathbf{W}} = \mathbf{v}^\top \mathbf{W}\mathbf{v} \in \mathbb{R}$ is the weighted 2-norm of vector $\mathbf{v}$ and w_i is a scalar to weight the total cost.

In practice, each phase has some goals associated with costs: the weights of these costs indicate how important it is to achieve these goals. Low weights are associated with regularization goals and high weights with task goals. Other variables to be explicitly specified are Δt and the duration of each phase, and hence its number of nodes. These time variables generally affect the convergence time of the algorithm that solves (3.1), which is described in Sect. 3.2.

3.1.1 Typical residuals

In the following we present a set of residuals useful to define tasks:

1. *State error*: A common way to specify a task is to set a desired state $\mathbf{x}^\star$ for a particular node. Thus, we can define a state residual as

$$\mathbf{r}_{\text{state},k} = \mathbf{x}^\star \ominus \mathbf{x}_k = \begin{bmatrix} \text{Log}(\mathbf{M}^{-1}_{B,k}\mathbf{M}^\star_B) \\ \mathbf{q}^\star_j - \mathbf{q}_{j,k} \\ \dot{\mathbf{q}}^\star_j - \dot{\mathbf{q}}_{j,k} \end{bmatrix} \tag{3.3}$$

The operator $\ominus$ stands for the difference between states expressed as a vector of the space tangent to the manifold of the state ([SDA18]): for elements in non-linear manifolds we have $\mathbf{Y} \ominus \mathbf{X} \triangleq \text{Log}(\mathbf{X}^{-1}\mathbf{Y})$, and in vector spaces simply $\mathbf{y} \ominus \mathbf{x} \triangleq \mathbf{y} - \mathbf{x}$. This allows us to define residuals involving only a part of the state, e.g.,

$$\begin{aligned} \mathbf{r}_{\text{pos},k} &= \mathbf{p}^\star - \mathbf{p}_k && \text{position} \\ \mathbf{r}_{\text{ori},k} &= \mathbf{R}^\star \ominus \mathbf{R}_k && \text{orientation} \\ \mathbf{r}_{\text{linv},k} &= \mathbf{v}^\star - \mathbf{v}_k && \text{lin. vel.} \\ \mathbf{r}_{\text{angv},k} &= \boldsymbol{\omega}^\star - \boldsymbol{\omega}_k && \text{ang. vel.} \end{aligned}$$

2. *Control error*: Similarly to the state residual, we can define a control residual as

$$\mathbf{r}_{\text{control},k} = \mathbf{u}^\star - \mathbf{u}_k \,.$$

Normally, this residual is used as a regularization cost, taking into account $\mathbf{u}^\star = \mathbf{0}$ and giving it a low weight.

3. *Pose and velocity of frames*: It is a common practice in robotics to add a desired task in the operational space, e.g. a desired pose for the end-effector. It may also be useful to define desired poses for any arbitrary frame $\mathcal{F}$ in the robot. Thus,

$$\mathbf{r}_{\text{pose},k} = \mathbf{M}^*_{\mathcal{F}} \ominus \mathbf{M}_{\mathcal{F},k} = \text{Log}(\mathbf{M}^{-1}_{\mathcal{F},k}\mathbf{M}^*_{\mathcal{F}}) \,,$$

where $\mathbf{M}^*_{\mathcal{F}}$, $\mathbf{M}_{\mathcal{F},k} \in SE(3)$ are the desired pose and the pose of frame $\mathcal{F}$ at node k, respectively. We can also specify a desired linear and/or angular velocity for a frame $\mathcal{F}$ at node k, i.e.,

$$\mathbf{r}_{\text{velocity},k} = \mathbf{v}^*_{\mathcal{F}} - \mathbf{v}_{\mathcal{F},k} \,.$$

As before, we can define residuals involving only parts (position, orientation, etc.) of the frame.

4. *Contact cone*: When defining the equation of motion 2.2 we assume zero velocity at the contact points. However, depending on the direction of the contact force and the surface characteristics, slippage may occur. To avoid this, we constrain the contact forces to be inside the Coulomb's dry friction cone, which is defined by the coefficient of friction μ between the contact point and the surface. For computational purposes, we approximate the friction cone by a square pyramid, i.e., we impose the conditions

$$|f_{c,x}| \leq \mu f_{c,z} \,, \tag{3.4a}$$

$$|f_{c,y}| \leq \mu f_{c,z} \,, \tag{3.4b}$$

$$f_{c,z} > 0 \,, \tag{3.4c}$$

where $\mathbf{f}_c = [f_{c,x}\; f_{c,y}\; f_{c,z}]^\top$ is the contact force expressed in the reference frame of the contact surface. Notice that we can express (3.4) in matrix form,

$$\mathbf{A}\mathbf{f}_c = \begin{bmatrix} 1 & 0 & -\mu \\ -1 & 0 & -\mu \\ 0 & 1 & -\mu \\ 0 & -1 & -\mu \\ 0 & 0 & -1 \end{bmatrix} \mathbf{f}_c \leq \mathbf{0}\,,$$

where we use component-wise inequality. In this thesis we are considering a solver that cannot handle this kind of hard constraints. We therefore add them as soft constraints in the cost function by means of a quadratic barrier, i.e., through a residual whose five components are assigned according to

$$r^i_{\text{contact},k} = \begin{cases} 0 & \text{if } \mathbf{a}^i\mathbf{f}_c \leq 0, \\ \mathbf{a}^i\mathbf{f}_c & \text{if } \mathbf{a}^i\mathbf{f}_c > 0, \end{cases} \quad \text{for } i = 1, \dots, 5,$$

where $\mathbf{a}^i$ represents the i-th row of the matrix $\mathbf{A}$ above. A reasonable value for μ is around 0.7.

3.2 Differential Dynamic Programming

Differential Dynamic Programming (DDP) is a low-complexity algorithm for solving fastly (3.1). DDP takes advantage of the inherently sequential structure of the OCP: past control inputs affect future states, but future control inputs cannot affect past states. The consequence of such a Markovian structure is an inherent sparsity exploited by DDP: instead of factorizing a single large matrix, it performs multiple factorizations of many small matrices.

DDP is an approximation of the Dynamic Programming (DP) algorithm, i.e. it uses a second-order Taylor approximation of the cost-to-go function and combines it with Newton's method. It starts with a non-optimal trajectory for the controls $\mathbf{U} \triangleq \mathbf{u}_{0:N-1}$ and states $\mathbf{X} \triangleq \mathbf{x}_{0:N}$ and iteratively computes small improvements.

Each iteration of the DDP algorithm consists of two main steps or passes, which run over all the nodes (discretization points) of the trajectory: the backward pass and the forward pass. In the following we give the main formulation involved in these passes.

1. *Backward pass*: This pass results in an optimal policy at every node. We start the derivation of the equations involved in the backward pass by expressing the cost associated with the tail of the trajectory (from any node i to the terminal node N), i.e.,

$$J_i(\mathbf{x}_i, \mathbf{U}_i) = \sum_{k=i}^{N-1} \ell_k(\mathbf{x}_k, \mathbf{u}_k) + \ell_N(\mathbf{x}_N)\,.$$

Thanks to the Bellman principle, we can recursively solve this problem as

$$\begin{aligned} V_i(\mathbf{x}_i) &= \min_{\mathbf{u}_i} Q_i(\mathbf{x}_i, \mathbf{u}_i) \\ &= \min_{\mathbf{u}_i}[\ell_i(\mathbf{x}_i, \mathbf{u}_i) + V_{i+1}(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i))] \,, \end{aligned}$$

$Q_i(\mathbf{x}_i, \mathbf{u}_i)$ being the action-value function and $V_i(\mathbf{x}_i)$ the cost-to-go function. For the sake of clarity, in the following the subindices i are omitted and the cost-to-go at the next time step is shown with the prime symbol, i.e., $V'(\mathbf{f}(\mathbf{x}, \mathbf{u})) \triangleq V_{i+1}(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i))$. Expanding $Q(\mathbf{x}, \mathbf{u})$ up to the second order we get

$$Q(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}) \approx Q(\mathbf{x}, \mathbf{u}) + \frac{1}{2} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{q}_\mathbf{x}^\top & \mathbf{q}_\mathbf{u}^\top \\ \mathbf{q}_\mathbf{x} & \mathbf{Q}_\mathbf{xx} & \mathbf{Q}_\mathbf{xu} \\ \mathbf{q}_\mathbf{u} & \mathbf{Q}_\mathbf{xu}^\top & \mathbf{Q}_\mathbf{uu} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix} \,,$$

where

$$\mathbf{q}_\mathbf{x} = \boldsymbol{\ell}_\mathbf{x} + \mathbf{f}_\mathbf{x}^\top \mathbf{V}'_\mathbf{x} \,, \tag{3.5}$$

$$\mathbf{q}_\mathbf{u} = \boldsymbol{\ell}_\mathbf{u} + \mathbf{f}_\mathbf{u}^\top \mathbf{V}'_\mathbf{x} \,, \tag{3.6}$$

$$\mathbf{Q}_\mathbf{xx} = \boldsymbol{\ell}_\mathbf{xx} + \mathbf{f}_\mathbf{x}^\top \mathbf{V}'_\mathbf{xx} \mathbf{f}_\mathbf{x} + \mathbf{V}'_\mathbf{x} \cdot \mathbf{f}_\mathbf{xx} \,, \tag{3.7}$$

$$\mathbf{Q}_\mathbf{ux} = \boldsymbol{\ell}_\mathbf{ux} + \mathbf{f}_\mathbf{u}^\top \mathbf{V}'_\mathbf{xx} \mathbf{f}_\mathbf{x} + \mathbf{V}'_\mathbf{x} \cdot \mathbf{f}_\mathbf{ux} \,, \tag{3.8}$$

$$\mathbf{Q}_\mathbf{uu} = \boldsymbol{\ell}_\mathbf{uu} + \mathbf{f}_\mathbf{u}^\top \mathbf{V}'_\mathbf{xx} \mathbf{f}_\mathbf{u} + \mathbf{V}'_\mathbf{x} \cdot \mathbf{f}_\mathbf{uu} \,, \tag{3.9}$$

where $\mathbf{V}'_\mathbf{x}$, $(\boldsymbol{\ell}_\mathbf{x}, \boldsymbol{\ell}_\mathbf{u})$, $(\mathbf{f}_\mathbf{x}, \mathbf{f}_\mathbf{u})$ and $\mathbf{V}'_\mathbf{xx}$, $(\boldsymbol{\ell}_\mathbf{xx}, \boldsymbol{\ell}_\mathbf{ux}, \boldsymbol{\ell}_\mathbf{uu})$, $(\mathbf{f}_\mathbf{xx}, \mathbf{f}_\mathbf{ux}, \mathbf{f}_\mathbf{uu})$, describe the Jacobians and Hessians of the cost-to-go, cost and dynamics functions, respectively. By plugging the expansion into the Bellman equation we minimize the quadratic approximation of the cost-to-go function and so the decision variable becomes $\delta\mathbf{u}$. Minimizing with respect to $\delta\mathbf{u}$ yields to the optimal policy

$$\delta\mathbf{u}^*(\delta\mathbf{x}) = \mathbf{k} + \mathbf{K}\delta\mathbf{x}$$

with $\mathbf{k} \triangleq -\mathbf{Q}_\mathbf{uu}^{-1}\mathbf{q}_\mathbf{u}$ and $\mathbf{K} \triangleq -\mathbf{Q}_\mathbf{uu}^{-1}\mathbf{Q}_\mathbf{ux}$. If we insert this optimal policy into the quadratic approximation of the cost-to-go function, it becomes a function only of $\delta\mathbf{x}$:

$$V(\mathbf{x} + \delta\mathbf{x}) \approx V(\mathbf{x}) + \mathbf{p}^\top \delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^\top \mathbf{P} \delta\mathbf{x}$$

where

$$\begin{aligned} \mathbf{p} &= \mathbf{q}_\mathbf{x} + \mathbf{K}^\top \mathbf{q}_\mathbf{u} + \mathbf{K}^\top \mathbf{Q}_\mathbf{uu}\mathbf{k} + \mathbf{Q}_\mathbf{xu}\mathbf{k} \\ \mathbf{P} &= \mathbf{Q}_\mathbf{xx} + \mathbf{K}^\top \mathbf{Q}_\mathbf{uu}\mathbf{k} + \mathbf{Q}_\mathbf{xu}\mathbf{K} + \mathbf{K}^\top \mathbf{Q}_\mathbf{ux} \end{aligned}$$

To sum up, these are the steps of the backward pass:

```
$V_N(\mathbf{x}_N) \leftarrow \ell_N(\mathbf{x}_N)$
$\mathbf{p}_N \leftarrow \nabla_{\mathbf{x}}\ell_N(\mathbf{x}_N)$
$\mathbf{P}_N \leftarrow \nabla_{\mathbf{xx}}\ell_N(\mathbf{x}_N)$
for $i \leftarrow N-1$ to $0$ do
    $V_i(\mathbf{x}_i + \delta\mathbf{x}_i) = \min_{\delta\mathbf{u}_i} Q_i(\mathbf{x}_i, \mathbf{u}_i)$
    $\delta\mathbf{u}_i \leftarrow \mathbf{k}_i + \mathbf{K}_i\delta\mathbf{x}_i$
    $\mathbf{p}_i \leftarrow \mathbf{q}_{\mathbf{x}i} + \mathbf{K}_i^\top\mathbf{q}_{\mathbf{u}i} + \mathbf{K}_i^\top\mathbf{Q}_{\mathbf{uu}i}\mathbf{k}_i + \mathbf{Q}_{\mathbf{xu}i}\mathbf{k}_i$
    $\mathbf{P}_i \leftarrow \mathbf{Q}_{\mathbf{xx}i} + \mathbf{K}_i^\top\mathbf{Q}_{\mathbf{uu}i}\mathbf{K}_i + \mathbf{Q}_{\mathbf{xu}i}\mathbf{K}_i + \mathbf{K}_i^\top\mathbf{Q}_{\mathbf{ux}i}$
end for
```

2. *Forward pass*: Given a current guess of the trajectories $(\mathbf{X}, \mathbf{U})$ and the optimal policies computed during the backward pass, the forward pass applies appropriate modifications to them, node by node and proceeding forward. $\hat{\mathbf{x}}$ and $\hat{\mathbf{u}}$ are the updated values for the states and controls, respectively. This procedure is also known as the *non-linear rollout.* The parameter $\alpha \in (0, 1]$ indicates the length of the step taken by the current iteration. A value of $\alpha = 1$ results in the application of a full step.

```
$\hat{\mathbf{x}}_0 \leftarrow \mathbf{x}_0$
for $i \leftarrow 0$ to $N-1$ do
    $\hat{\mathbf{u}}_i \leftarrow \mathbf{u}_i + \alpha\mathbf{k}_i + \mathbf{K}_i(\hat{\mathbf{x}}_i - \mathbf{x}_i)$
    $\hat{\mathbf{x}}_{i+1} \leftarrow \mathbf{f}(\hat{\mathbf{x}}_i, \hat{\mathbf{u}}_i)$
end for
```

In its original form, DDP has poor globalization capabilities, i.e., from an arbitrary initial guess it struggles to converge to a good optimum. This is mainly due to the fact that DDP does not allow for infeasible trajectories during the optimization process, i.e., the constraints $\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)$ must always be satisfied, as in a single shooting approach. A variant of the standard DDP is a solver called Feasibility-prone Differential Dynamic Programming (FDDP), whose numerical behavior is similar to the classical multiple shooting approach. FDDP significantly improves the globalization capability and the convergence rate of the solver.

Another limitation of the original DDP is its inability to handle constraints other than those imposed by the system dynamics. In the robotics community, one of the first successful methods for incorporating inequality constraints into DDP is known as Box-DDP. It focuses on handling control limits during the computation of the backward pass, i.e., in the minimization of the action-value function. A recent work [Mas+22] proposed a feasibility-driven search for nonlinear optimal control problems with control limits, called Box-FDDP, which is the one used in this thesis. Except for the control limits and dynamics, it handles all remaining constraints (e.g., state and friction cone) through quadratic penalization.

3.3 Optimal trajectories generation

The software libraries used to generate optimal trajectories are Pinocchio [PIN] and Crocoddyl [CRO]. Both are open-source, mostly written in C++ with Python bindings. Pinocchio is a library for efficiently computing the dynamics (and derivatives) of a robot model, or of any articulated rigid-body model. It implements the classical algorithms following the methods described in [Fea08]. Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. Crocoddyl computes optimal trajectories along with optimal feedback gains. It uses Pinocchio for fast computation of robots dynamics and their analytical derivatives. It is one of the most efficient libraries for computing the optimal control with particular enphasis to contact dynamics.

A Python script using Pinocchio and Crocoddyl implementing the Box-FDDP solver was developed to generate the optimal state and control trajectories $\mathbf{X}^\star = \{\mathbf{x}_0^\star, \dots, \mathbf{x}_N^\star\}$ and $\mathbf{U}^\star = \{\mathbf{u}_0^\star, \dots, \mathbf{u}_{N-1}^\star\}$, which are then used as the reference state and feedforward terms for the low-level impedance controller in (2.8). In practice, these are matrices with dimensions of $N \times (2n_v + 1)$ and $(N-1) \times n_j$ respectively, the k-th row being associated with the time step k. Gepetto-viewer is used to visualize the resulting motion of the robot, while a C++ script has been developed to convert the trajectories from the Python data structure `numpy array` to the `Eigen::MatrixXd` C++ data structure. This conversion is necessary to work with the ROS2 application, described in Chap. 4, which allows the trajectories to be executed on the real robot.

The URDF model of Solo-12 is contained in a repository called `example-robot-data` [EXA]. For the trajectories generation, a modified version has been used corresponding to the initial hardware configuration shown in Table 1.1. The only modification to the original model is the addition of the power distribution module, which adds 0.3 kg to the entire weight of the robot, shifting the center of mass backward with respect to the geometric center of the base link (Fig. 3.2). An estimate of the power distribution moments of inertia was obtained by approximating the power distribution module to a parallelepiped and using the formulas $I_{xx} = \frac{m}{12}(y^2 + z^2)$, $I_{yy} = \frac{m}{12}(x^2 + z^2)$, $I_{zz} = \frac{m}{12}(x^2 + y^2)$.

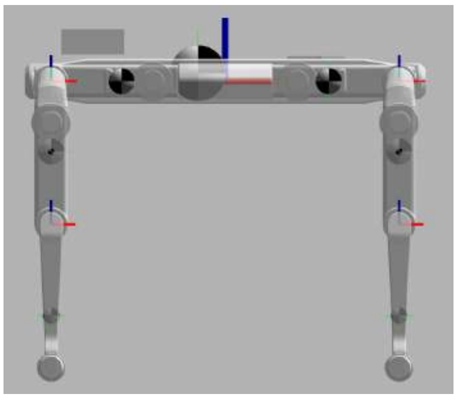


Figure 3.2: Centers of mass of the links

3.4 Jump

The jump trajectory can be divided into 3 main phases:

1. *Preparation*: the purpose of this phase is to accelerate the base to leave the ground with a linear velocity high enough to reach the desired height of the jump $z^\star$. If $z^\star$ is large enough, the quadruped will lower its base before the takeoff;
2. *Flying*: once the robot has jumped, the motion is governed only by the gravity;
3. *Landing*: the purpose of this phase is to avoid abrupt impact with the ground while decelerating the robot. To this end, a suitable impedance must be imposed.

The tasks of the jump are to reach $z^\star$ at the apex of the ballistic trajectory, keep the base orientation horizontal, retract the legs to land properly, and zero joints velocities at the end of the trajectory. Since the goal is to reach $z^\star$, from the OCP perspective only the first two phases are considered. The landing phase is the most delicate because an abrupt landing can cause damage to the robot's hardware, and especially to the HFE and KFE joints. This issue will be discussed in the Chap. 4.

In Fig. 3.3 we show the phases of the jump in an OCP with $z^\star = 0.6$ m, $\Delta t = 2$ ms and torque limits of ± 2 Nm, while Figg. 3.5, 3.6 and 4.3 show the joints positions, velocities, torques and base height.

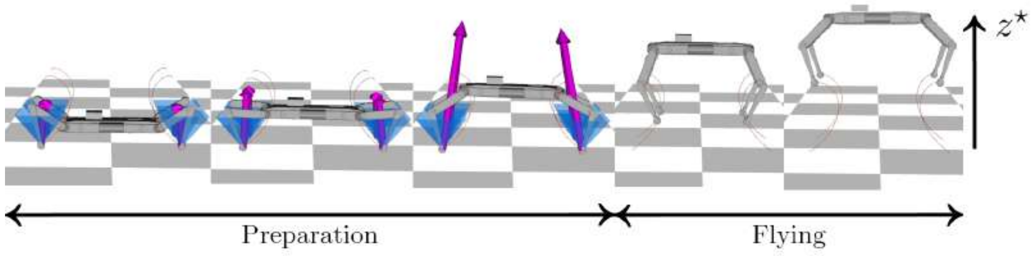


Figure 3.3: Jump phases

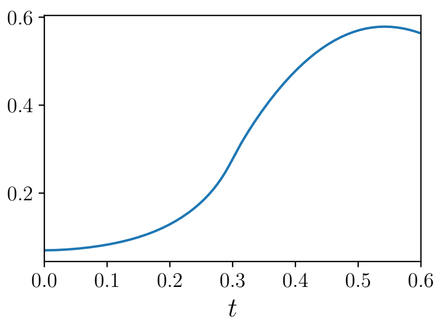


Figure 3.4: Base height (jump)

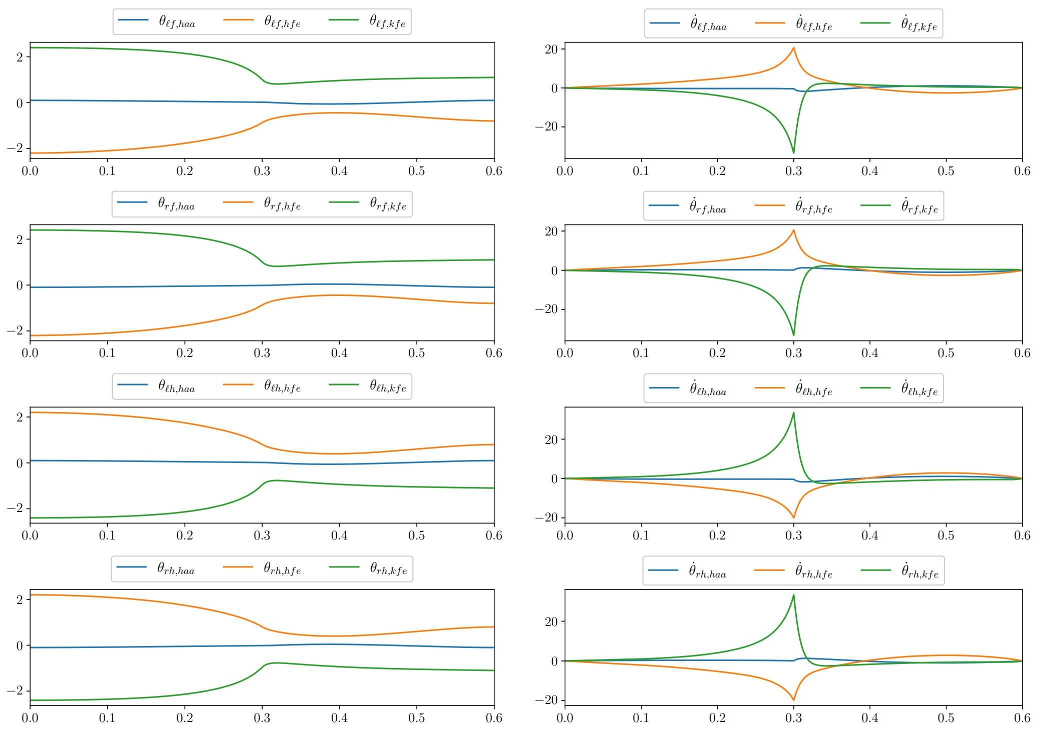


Figure 3.5: Optimal joints positions (left) and velocities (right) for the jump

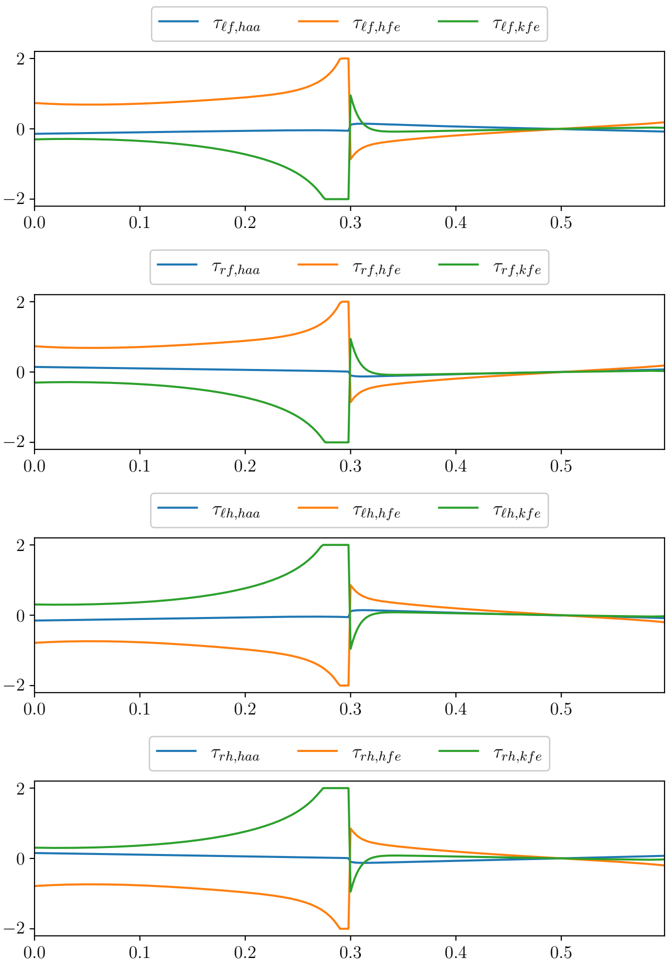


Figure 3.6: Optimal joints torques for the jump

3.5 Backflip

The backflip is a very agile maneuver, definitely more complex to perform than the jump since it involves jumping first with the front legs and then with the hind legs, so that the base link has a high enough pitch velocity to perform a 360° rotation in the flight phase, a situation in which the robot is underactuated. The motion phases are:

1. *Preparation*: the purpose of this phase is to prepare for the front legs jump;
2. *Jump with the front legs*: in this phase the robot jumps with the front legs so that the base link rotates about 100° around the pitch axis. This value has been chosen taking cue from the trajectories of backflip executed from quadrupeds of other research institutes;
3. *Flying*: in this phase the robot spins in the air while retracting the legs to be ready for the landing configuration;
4. *Landing*: as the hind legs spin faster than the front ones, they need to absorb much more energy. This must be accomplished by selecting an asymmetrical configuration of front and hind joint angles and impedances.

In Fig. 3.7 we show the phases of the backflip in an OCP with $\Delta t = 2$ ms and torque limits of ± 2 Nm, while Figg. 3.8 and 3.9 show the joints positions, velocities and torques.

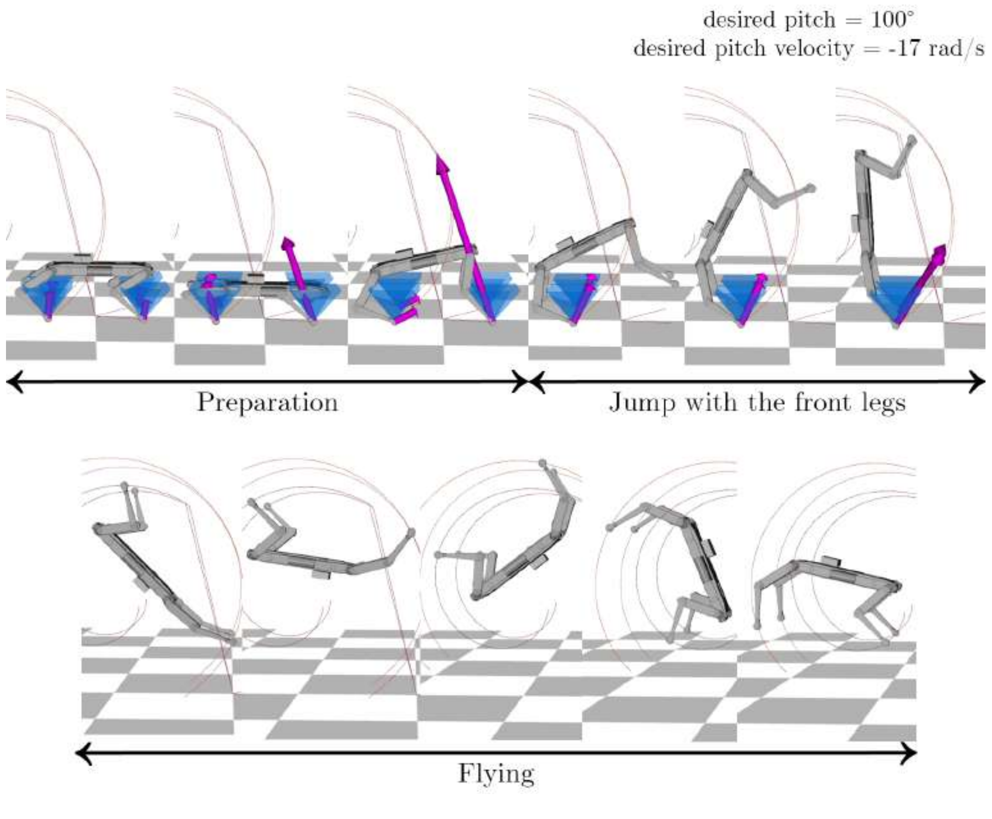


Figure 3.7: Backflip phases

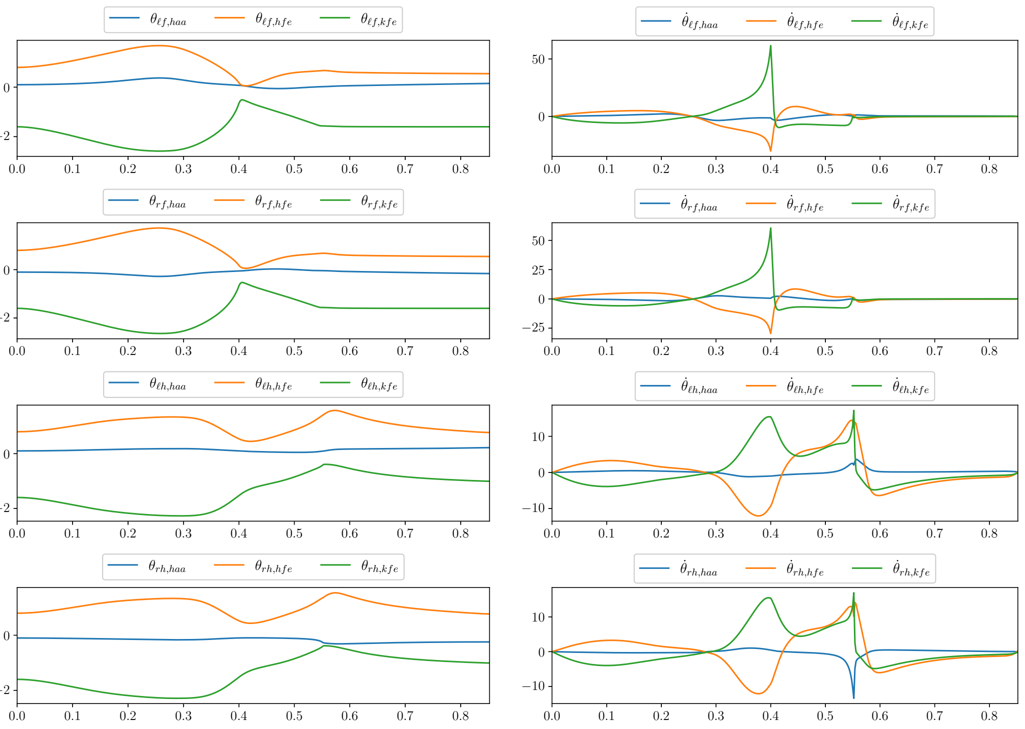


Figure 3.8: Optimal joints positions (left) and velocities (right) for the backflip

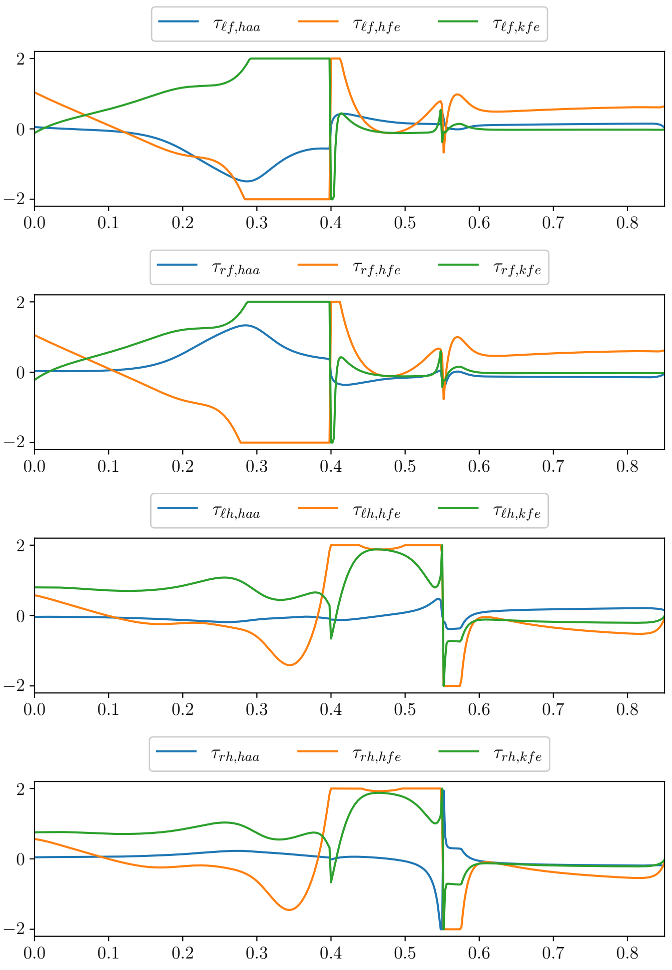


Figure 3.9: Optimal joints torques for the backflip

4. Implementation Results

In this chapter we will present the results obtained in simulation with PyBullet simulator [CB22] and on the real robot using ROS2.

4.1 Simulation

PyBullet is a fast and easy to use Python module for robotics simulation and machine learning, with a focus on sim-to-real transfer. With PyBullet one can load articulated bodies from URDF, SDF, MJCF and other file formats. PyBullet provides forward dynamics simulation, inverse dynamics computation, forward and inverse kinematics, collision detection and ray intersection queries.

A Python script using Pinocchio and PyBullet APIs was developed to simulate the optimal trajectories shown in Chap. 3 and the low-level impedance controller (2.8). Fig. 4.1 shows PyBullet graphical user interface.

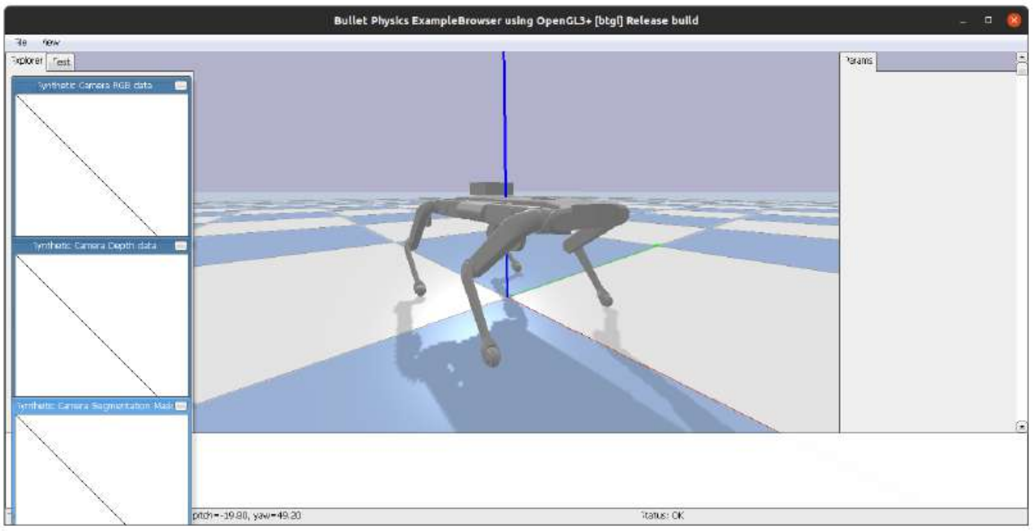

Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build
File
View
Explorer
Test
Params
Synthetic Camera RGB data
Synthetic Camera Depth data
Synthetic Camera Segmentation Mask
pitch=-19.80, yaw=49.20
Status: OK

Figure 4.1: PyBullet GUI

4.1.1 Jump

To simulate the trajectory shown in Sect. 3.4, PD gains values must be chosen for each phase of the motion. k_P must be high enough to ensure good tracking of the state trajectory, but low enough not to reach often the torque saturation. k_D must be high enough to ensure sufficient damping. We chose the following values:

	Preparation	Flying	Landing
k_P	2	1	3.0
k_D	0.3	0.1	0.2

Table 4.1: PD gains for the jump simulation

In Fig. 4.2 we show the simulation of the jump, while in Figg. 4.4 and 4.5 we show joints positions, velocities and the torques of the left-front leg (each leg behaves qualitatively the same because of the motion's symmetry). Plots are segmented into three time intervals with vertical lines: in $t \in [0, t_0)$ the robot stays put in the initial configuration, in $t \in [t_0, t_2)$ it executes the optimal trajectory and in $t \geq t_2$ it lands and impacts the ground. The main difference with the trajectories shown in Figg. 3.5 and 3.6 is the landing phase: in $t = t_3$ the robot lands on the ground, causing a spike in joint velocities and torques to dampen the impact and a steady-state error in joints positions.

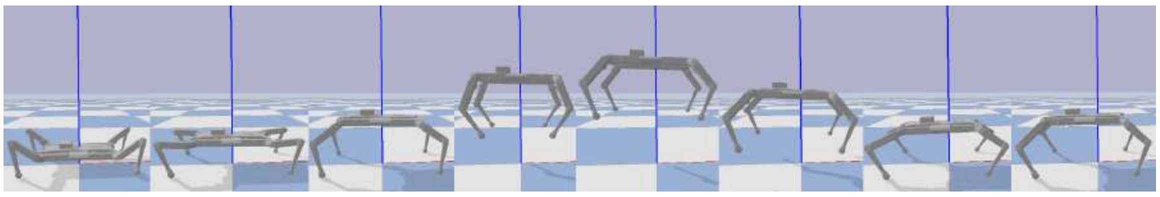


Figure 4.2: Jump simulation

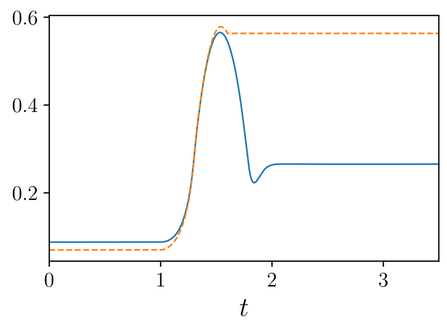
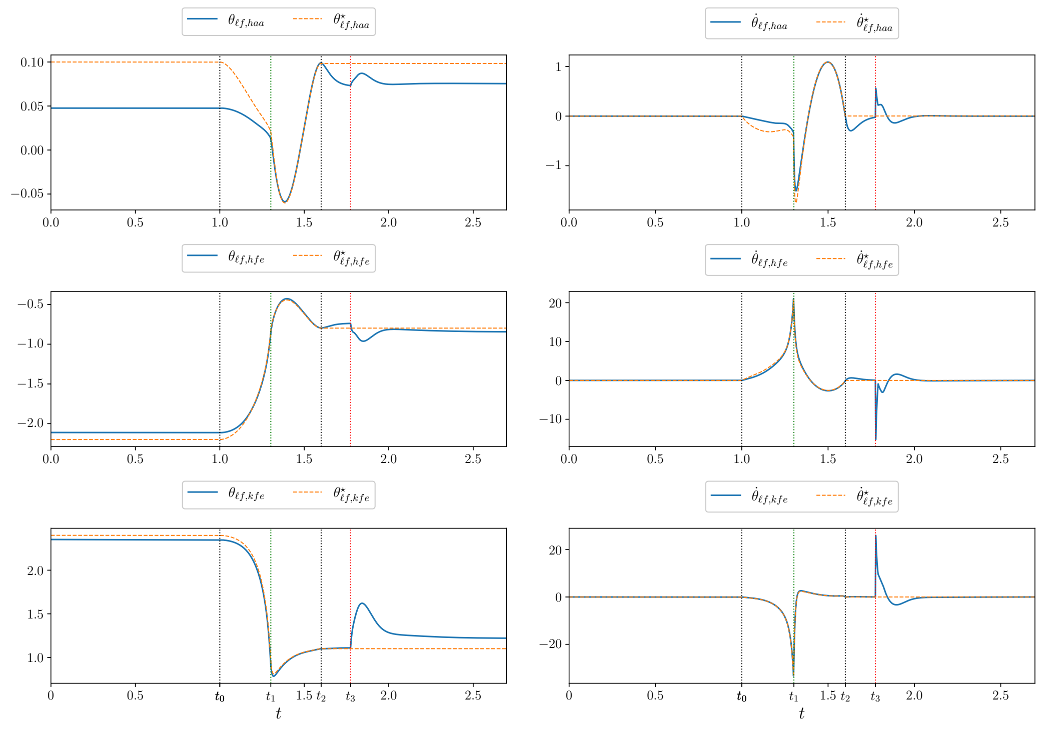


Figure 4.3: Base height (jump simulation). Full line: actual height, dashed line: reference height



(a) Joints positions

(b) Joints velocities

Figure 4.4: Joints positions and velocities (jump). Full line: actual values, dashed line: reference values. The black vertical lines delimit the beginning and the end of the optimal trajectory, green lines correspond to contact loss, red lines to contact gain. $t_0 = 1$ s: start of the trajectory; $t_1 = 1.302$ s: loss of contact; $t_2 = 1.6$ s: end of the trajectory; $t_3 = 1.774$ s: impact with the ground.

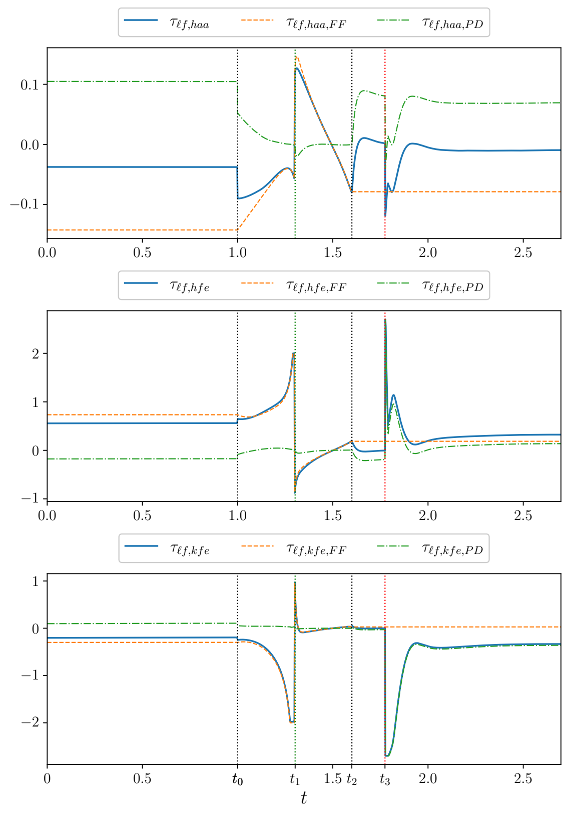


Figure 4.5: Left-front leg torques (jump). Dashed line: feedforward torque, dash-dot line: PD action, full line: total torque.

4.1.2 Backflip

The gains chosen for the backflip are:

	Preparation	Flying	Landing
k_P	5	5	5
k_D	0.3	0.1	0.3

Table 4.2: PD gains for the backflip simulation

In Fig. 4.6 we show the simulation of the backflip, while in Figg. 4.7, 4.8 and 4.9 we show joints positions, velocities and torques for each joint. For a safe landing, the final joints configuration has been chosen such that the support polygon is wider than that of the initial configuration: this prevents the elbow joints from impacting the ground and the robot from falling to the side. As we can see in Fig. 4.9 the impact causes chattering in the torques.

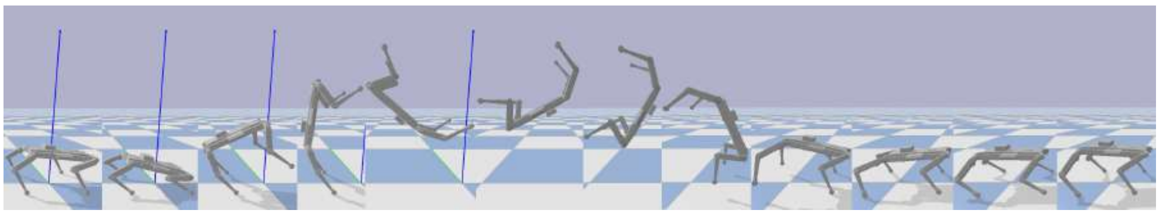


Figure 4.6: Backflip simulation

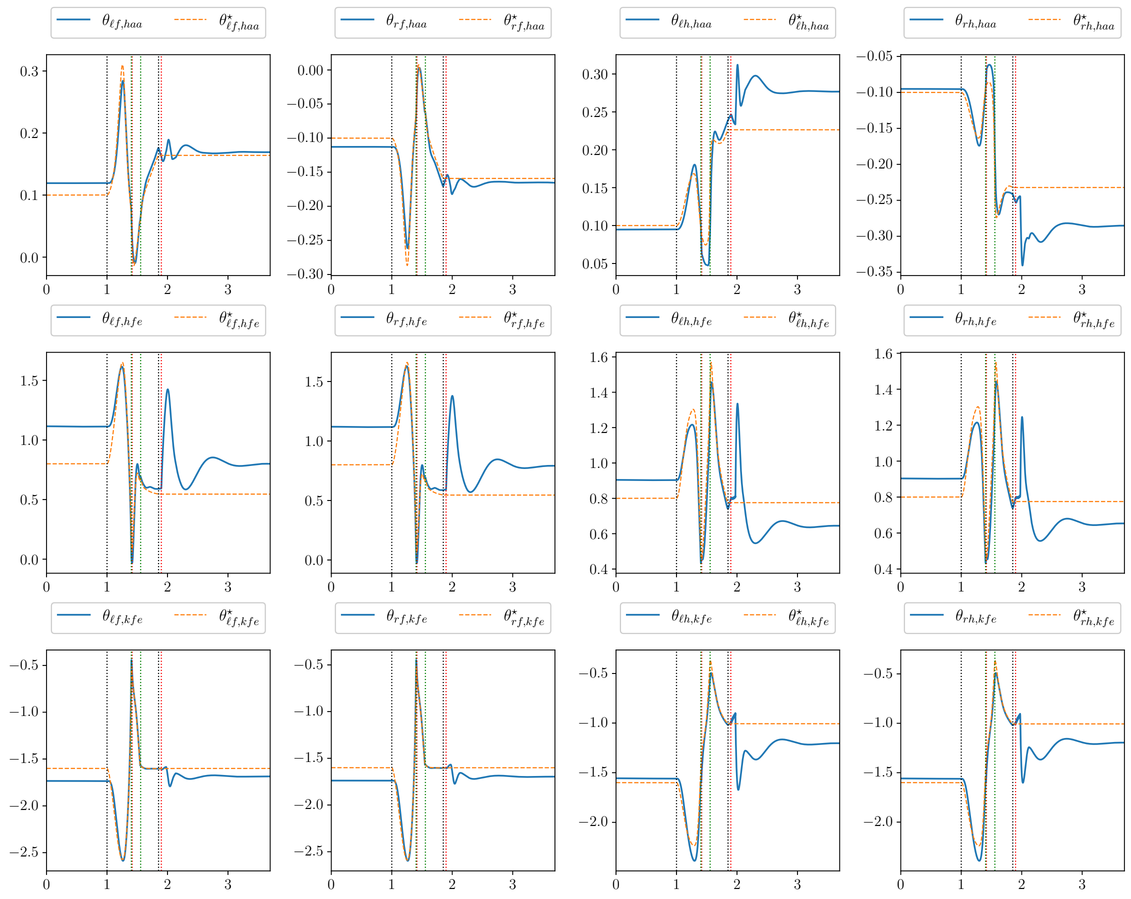


Figure 4.7: Joints positions (backflip). Full line: actual values, dashed line: reference values

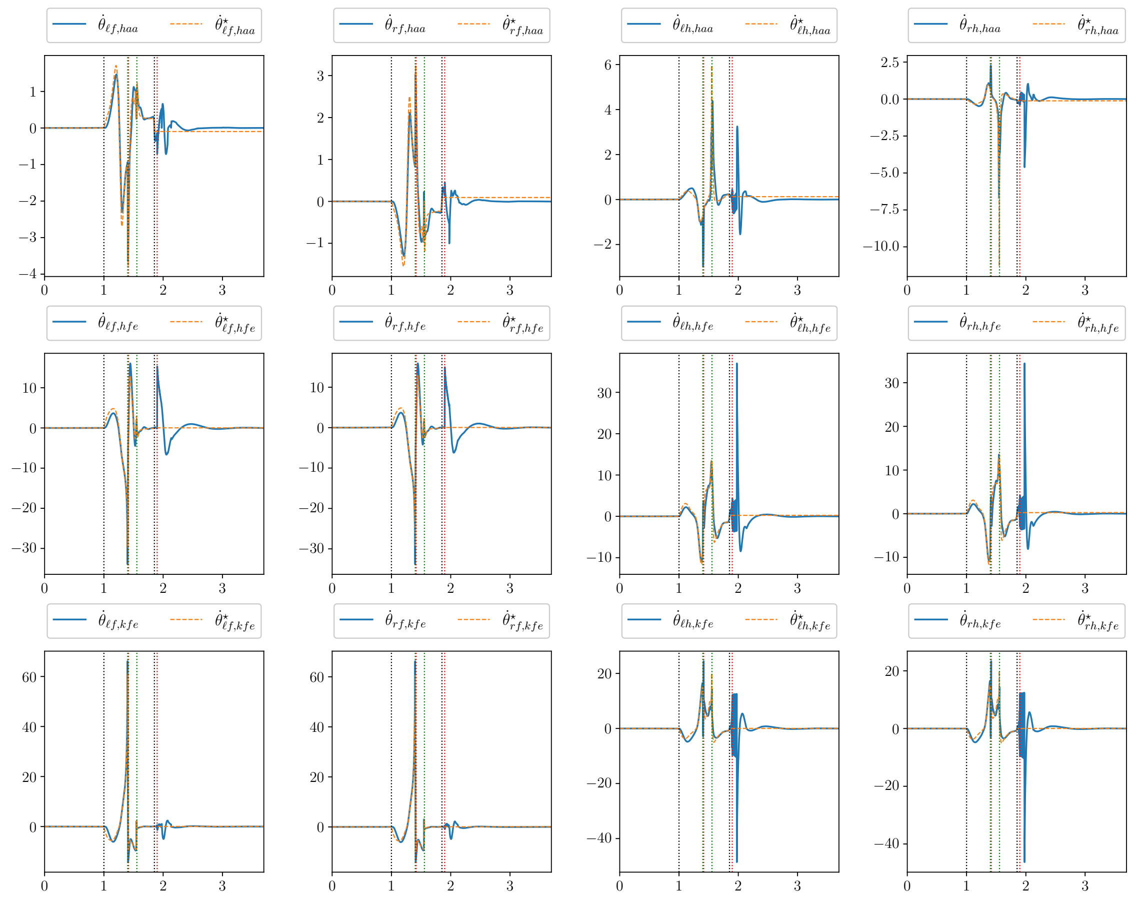


Figure 4.8: Joints velocities (backflip). Full line: actual values, dashed line: reference values

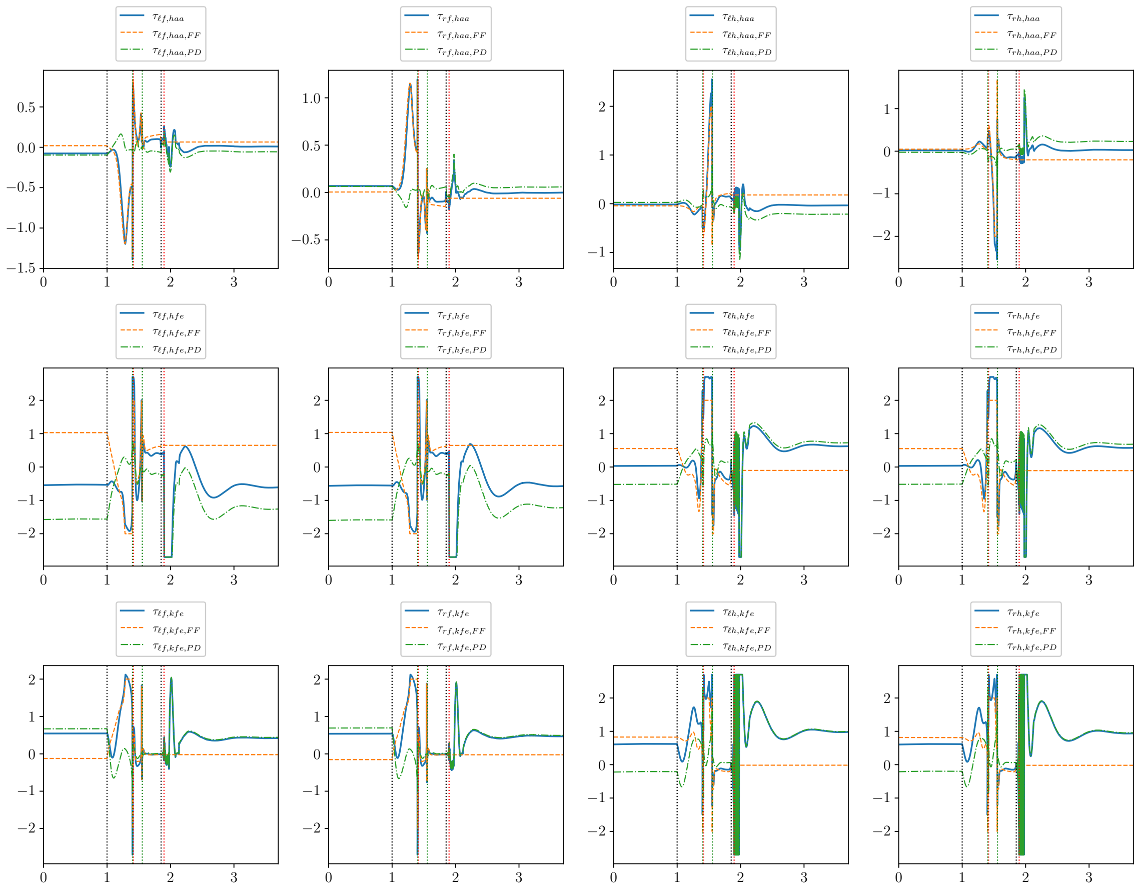


Figure 4.9: Joints torques (backflip). Dashed line: feedforward torque, dash-dot line: PD action, full line: total torque.

4.2 ROS2 Application

In order to test the optimal trajectories on the real robot, a ROS2 application was developed in C++, which is an interface between the user and the low-level hardware interface. It consists of two ROS2 packages:

- *Offline Trajectory Publisher* (OTP): this package implements a finite state machine so that a user can test a trajectory by querying state transitions from a shell terminal;
- *Robot Interface* (ODRI): this package implements the ROS2 hardware interface of the robot.

In the preliminary phase the user requests ODRI `enable` and `start` transitions. In `Enabled` the motors are powered, encoders are ready and the robot goes into a default joint configuration; in `Running`, the motors accept commands via a ROS2 topic.

In the operational phase the user can play a given trajectory: in OTP `Enabled` the robot goes to the first joint configuration of the trajectory, then in `Running` the trajectory is executed. After the trajectory is executed, the user can play it again by going to `Enabled` through `restart`. If a problem occurs during the execution of the trajectory, the `Safe` state can be triggered which sets $\boldsymbol{\tau}^{\star} = \mathbf{0}$, $k_P = 0$ and $k_D > 0$ for a damped landing. The stateflows are shown in Fig. 4.10.

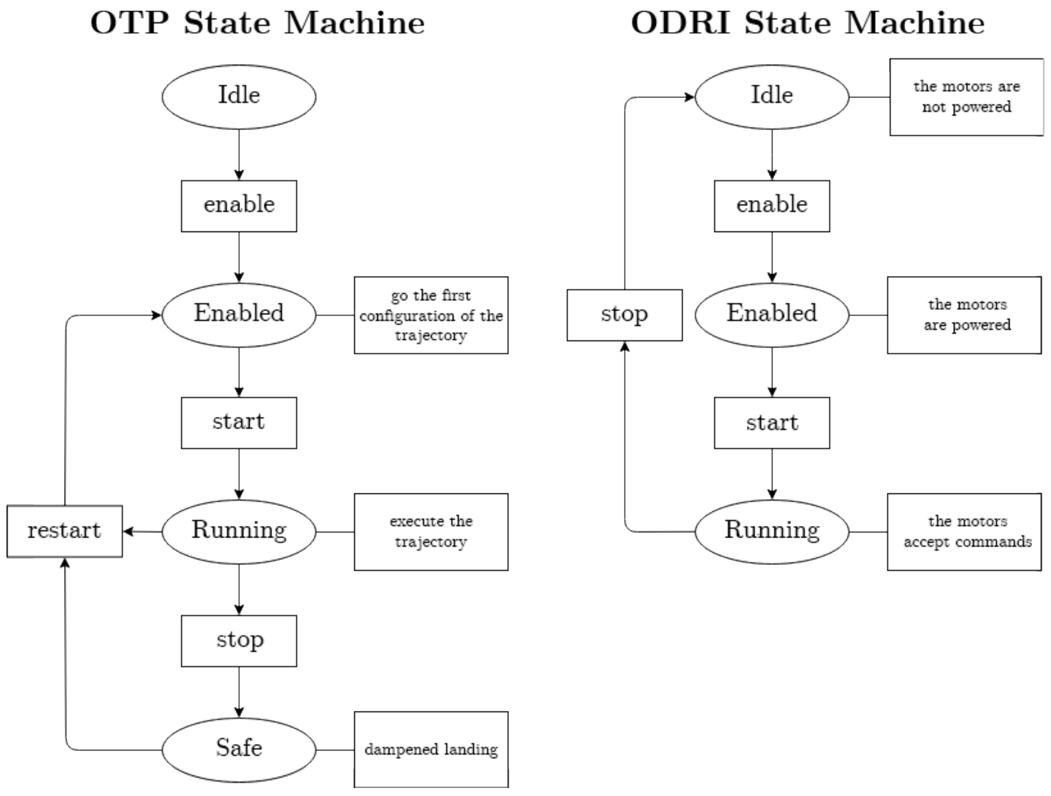


Figure 4.10: OTP and ODRI State Machines

4.3 Robot Calibration

When the robot is powered up, the joint angles are unknown, so the legs must be manually aligned before startup so that the joints are approximately at the zero position. The software moves all the joints and finds the closest encoder index position for each joint; when the index position is found the software adds an offset to the actual position and moves all the joints to the zero position (Fig. 4.11a).

For this procedure to work the offsets between the index positions and the zero joint positions must be measured and stored for each degree of freedom. The steps are:

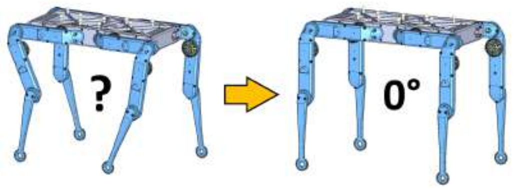
1. put the robot on the stand with the calibration tool (Fig. 4.11b);
2. turn on the robot;
3. remove the calibration tool and trigger ODRI `start_calibrating_offsets` transition from `Idle` to `Calibrating_offsets`: the joints will move slightly;
4. install again the calibration tool;
5. trigger ODRI `end_calibrating_offsets` transition from `Calibrating_offsets` to `Idle`;
6. save the offsets shown in the terminal to a configuration file.

If an actuator module is repaired, replaced, or modified, the offset between the index pulse and the zero joint position must be remeasured and stored.

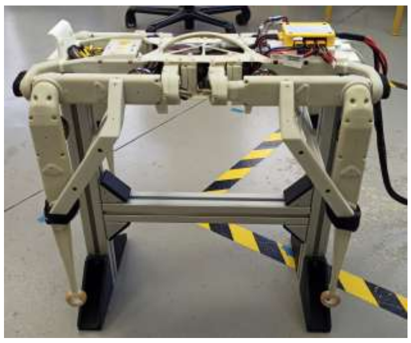
4.4 Experimental Procedure

The following procedure was used to perform experiments with the robot:

1. *Data preparation*: the Python files of the optimal state and control trajectories are generated and converted to a C++ compliant format;
2. *Trajectory loading*: the trajectories files are loaded into the OTP package;
3. *Robot preparation*: the robot is placed on a stand with its legs extended (Fig. 4.12);
4. *Test of the trajectory on the stand*: the trajectory is run with $k_{ff} = 0, k_P > 0$ and $k_D = 0$. This is needed to check if the reference joints positions $\mathbf{q}^\star$ are feasible. If so, we can proceed to the last step;
5. *Trajectory execution*: finally the trajectory is executed with $k_{ff} = 1, k_P > 0$ and $k_D > 0$. This step is repeated until a set of gains is found that produces satisfactory results.



(a) Zero configuration



(b) Calibration tool

Figure 4.11: Encoders calibration procedure

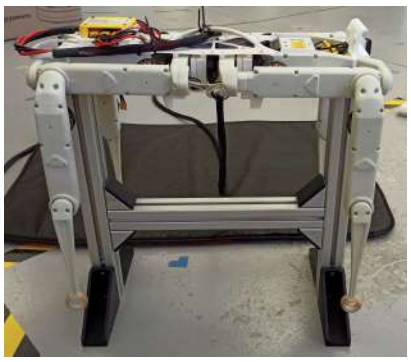


Figure 4.12: Solo-12 on the stand

When tuning the gains in (2.8) special attention must be paid to the current that powers the motors. Since the power supply can provide a maximum of 30 A, a saturation value i_{sat} must be set for the current of all the motors. If no saturation current is set for the motors, they may draw too much current and hence shut down to avoid damage. On the other hand, if the saturation value is too low, the feedforward torques cannot be tracked. Since each motor can deliver a maximum torque $\tau_{\text{max}} = 2.7$ Nm at 12 A, it makes sense to set $i_{\text{sat}} = 12$ A, but for safety reasons was chosen $i_{\text{sat}} = 8$ A for the jump and $i_{\text{sat}} = 10$ A for the backflip.

Another important aspect is the limits of the joints: for locomotion tasks it is reasonable to set limits so that the legs are never above the base link, but for agile maneuvers the joints need more freedom of motion, both for the preparation and landing phases. For the following experiments we set the following joints limits:

$$\begin{aligned}
\theta_{\ell f,haa},\ \theta_{rf,haa},\ \theta_{\ell h,haa},\ \theta_{rh,haa} &\in [-0.9,\ 0.9] \\
\theta_{\ell f,hfe},\ \theta_{rf,hfe} &\in [-2.4,\ 2.3] \\
\theta_{\ell h,hfe},\ \theta_{rh,hfe} &\in [-1.6,\ 2.6] \\
\theta_{\ell f,kfe},\ \theta_{rf,kfe},\ \theta_{\ell h,kfe},\ \theta_{rh,kfe} &\in [-3.1,\ 3.1]\,.
\end{aligned}$$

4.5 Jump

The goal of the experiment is to perform a jump, check the actual height reached by the base and land properly on the ground. The gains used for the experiment are

	Preparation	Flying	Landing
k_P	3	1	5
k_D	0.3	0.2	0.3

Table 4.3: PD gains for the jump

Fig. 4.13 shows the execution of the jump, while in Figg. 4.14, 4.15 we show joints positions, velocities and the torques of the left-front leg. The reached height is around 0.58 cm from the ground, which is quite close to what was specified in the OCP. Comparing Figg. 4.4 and Figg. 4.14 we can see that the tracking performance of the real robot is good compared to the simulated one. Only the HAA joints don't follow the references, but this is not a big deal because most of the work is done by the HFE and KFE joints. From Fig. 3.6 we can see that the reference torques go close to zero when the robot is in the air: as a result, the landing would be handled by PD action alone, potentially leading to an unstable final configuration. Thus, after executing the trajectory $\boldsymbol{\tau}^{\star}$ is set equal to the static torque calculated with (2.7) corresponding to the final configuration of the robot.

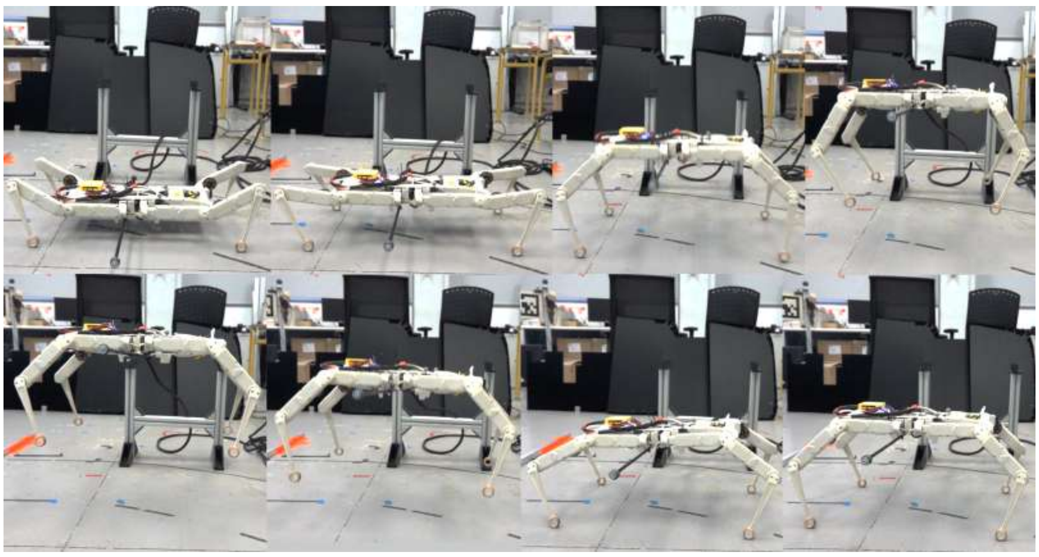
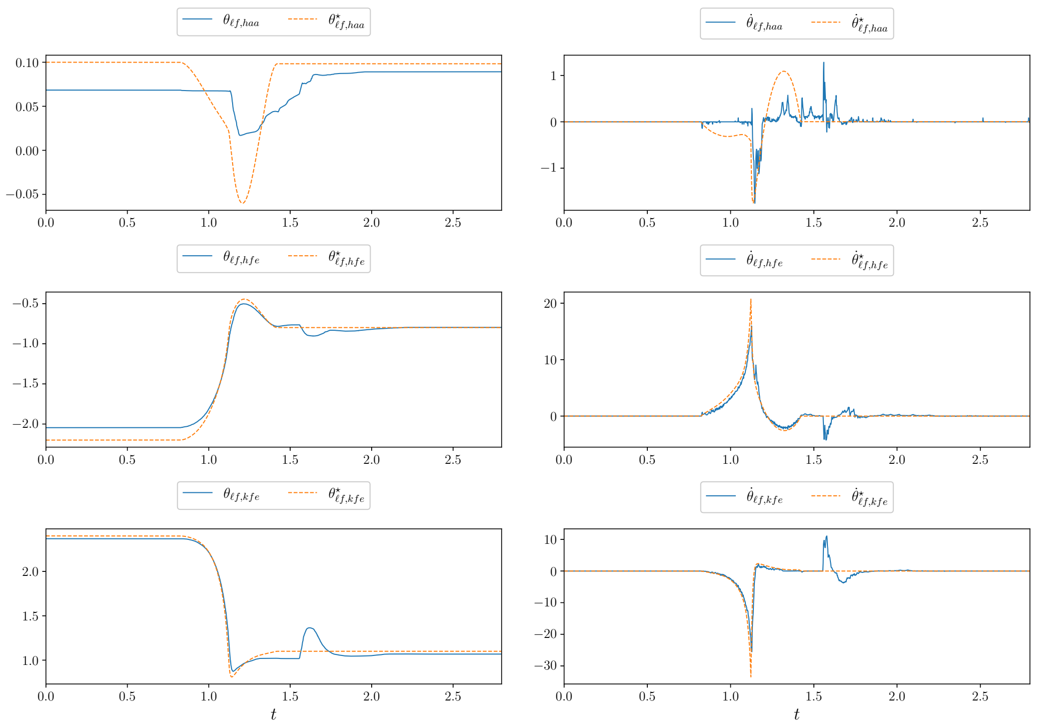


Figure 4.13: Jump execution



(a) Joints positions

(b) Joints velocities

Figure 4.14: Joints positions and velocities (jump). Full line: actual values, dashed line: reference values.

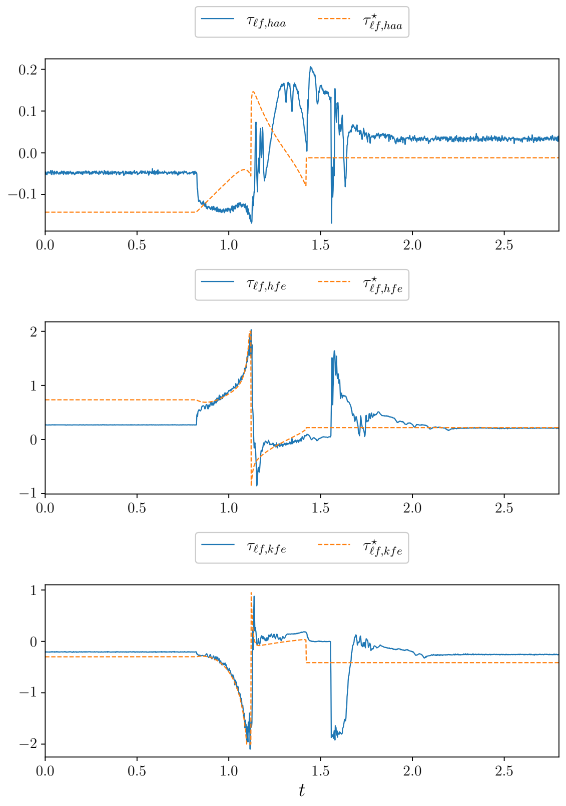


Figure 4.15: Left-front leg torques (jump). Dashed line: feedforward torque, full line: total torque.

4.6 Backflip

The shoulder and elbow joints are among the most fragile parts of the robot: in some experiments, the plastic parts shown in the upper right corner of Figure 1.1 b broke and had to be replaced with new 3D-printed parts, resulting in lost of time in reassembling and recalibrating the robot. This is why for the backflip experiments a system of cables was attached to the ceiling so that, by connecting it to the robot via a bar, it would allow to arrest its fall to prevent damage to the hardware.

The experiments did not produce a successful backflip. We can identify some possible causes:

- Although it works in simulation, the trajectory is open-loop because the controller gets feedback only from the encoders, but the reference motion is that of the base, not of the joints. It should also be noted that the dynamic model does not perfectly match the real robot;
- Modeling realistic contact between the foot and the ground is not trivial, so even including friction cone constraints into the OCP does not guarantee that slipping will not occur. In such cases, some of the energy provided by the motors will be dissipated in slipping rather than in jumping high, so that eventually the base doesn't rotate enough at the time of the impact.

Fig. 4.16 shows the best result achieved so far. A metal bar was put behind the back legs as a workaround to avoid slipping.

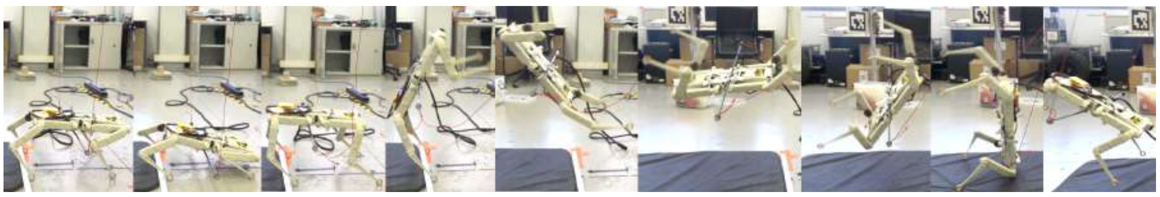


Figure 4.16: Backflip execution

5. Conclusions

In this thesis we explored the capabilities of the Solo-12 quadruped robot to perform agile motions using techniques based on Optimal Control. We began by defining motion trajectories, and then simulated and tested them on the real robot.

In principle, Optimal Control is easy to use because the generation of a trajectory is reasoned at a high level in terms of navigation and task phases. However, the more complex the trajectory, the more aspects must be taken into account to generate a physically feasible one, among all the durations of the phases and the weights to be assigned to the tasks, so at some point it becomes more art than science to make it work. The backflip trajectory required more than a month of coding and testing before it produced acceptable results in simulation.

Since Optimal Control is a model-based approach, it is crucial to have a dynamic model that is as realistic as possible for the simulation to produce results that are close to what we can expect to see with the real robot. This requires accurate modeling of each of the links in terms of mass and inertia in the robot's URDF file, which, due to time constraints, was only slightly modified to account for the total mass of the real robot.

Another critical aspect is the modeling of the contact loss/gain phases of the feet. In trajectory generation and visualization, the stance and swing phases of each foot are explicitly imposed by position and velocity constraints added to the foot frames, which does not account for physical problems such as slippage and, in particular, the fact that ground reaction forces can only be directed upward. To address these problems, it is important to include friction cone constraints in the Optimal Control Problem. However, these constraints did not prove helpful in the backflip experiments.

Overall, the results were satisfactory with respect to the objective of the thesis, also considering that the robot was recently acquired and little work had been done with it in the laboratory. The backflip trajectory obtained in this work could be used as a warm start for the solver to get a better result, along with a more management could potentially leading to a successful backflip maneuver.

Future projects could focus on the application of MPC techniques for locomotion, taking advantage of the hardware modifications described in Sect. 1.5.

Acknowledgements

Ringrazio il mio relatore per avermi messo in contatto con le persone dell'IRI con cui ho trascorso uno dei periodi più stimolanti e formativi della mia carriera universitaria. Vorrei ringraziare tutti i colleghi del laboratorio per la loro disponibilità e competenza, ed in particolare Joan Solà, Josep Martí-Saumell e Hugo Duarte per avermi supportato quotidianamente, e Joan López Zamora per i bei momenti e gli scambi di idee. Un ringraziamento speciale a ChatGPT per aver reso la mia vita da programmatore più semplice e a Maxime Klimecky del LAAS-CNRS per la parte di simulazione in PyBullet.

Sono grato a tutte le persone che mi hanno accompagnato in questo percorso, i colleghi di università vecchi e nuovi, gli amici e la mia famiglia. Ringrazio gli amici di Erasmus per le belle esperienze fatte insieme in questi mesi, che ricorderò sempre con affetto.

Merita una menzione speciale il team di supporto morale costituito dalla mia ragazza Alessandra, Miryam e mia sorella Arianna per avermi ascoltato nei momenti di difficoltà e incertezze. Sono molto fortunato che siate parte della mia vita.

I would like to thank my thesis supervisor for introducing me to the people at IRI, with whom I spent one of the most stimulating and formative periods of my university career. I would like to thank all of my colleagues at the lab for their helpfulness and expertise, and especially Joan Solà, Josep Martí-Saumell and Hugo Duarte for their daily support and Joan López Zamora for the good times and exchange of ideas. Special thanks to ChatGPT for making my programming life easier and to Maxime Klimecky from LAAS-CNRS for the simulation part in PyBullet.

I am grateful to all the people who have accompanied me on this journey, old and new university colleagues, friends and my family. I would like to thank the many people I had the pleasure to meet during my time in Barcelona for the wonderful experiences we had together during these months, which I will always remember fondly.

Bibliography

[CB22] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning.* `http://pybullet.org`. 2016–2022.

[CRO] CROCODDYL. *Crocoddyl Software Library.* `https://github.com/loco-3d/crocoddyl`.

[EXA] EXAMPLE-ROBOT-DATA. *example-robot-data repository.* `https://github.com/Gepetto/example-robot-data`.

[Fea08] Roy Featherstone. *Rigid Body Dynamics Algorithms.* Springer US, 2008. DOI: `10.1007/978-1-4899-7560-7`. URL: `https://doi.org/10.1007%5C%2F978-1-4899-7560-7`.

[Gri+20] F. Grimminger et al. "An Open Torque-Controlled Modular Robot Architecture for Legged Locomotion Research". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 3650–3657. DOI: `10.1109/LRA.2020.2976639`.

[KCK19] 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).* 2019, pp. 6295–6301. DOI: `10.1109/ICRA.2019.8793865`.

[Léz22] Pierre-Alexandre Léziart. "Locomotion control of a lightweight quadruped robot". Theses. UPS Toulouse, Oct. 2022. URL: `https://hal.laas.fr/tel-03936109`.

[Li+22] Chenhao Li et al. *Learning Agile Skills via Adversarial Imitation of Rough Partial Demonstrations.* 2022. arXiv: `2206.11693 [cs.RO]`.

[Mar+21] Josep Marti-Saumell et al. "Full-Body Torque-Level Non-linear Model Predictive Control for Aerial Manipulation". In: *CoRR* abs/2107.03722 (2021). arXiv: `2107.03722`. URL: `https://arxiv.org/abs/2107.03722`.

[Mas+22] Carlos Mastalli et al. *A Feasibility-Driven Approach to Control-Limited DDP.* 2022. arXiv: `2010.00411 [cs.RO]`.

[Neu+18] Michael Neunert et al. "Whole-Body Nonlinear Model Predictive Control Through Contacts for Quadrupeds". In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 1458–1465. DOI: `10.1109/LRA.2018.2800124`.

[ODR] ODRI. *Open Dynamic Robot Initiative.* `https://open-dynamic-robot-initiative.github.io/`.

[PIN] PINOCCHIO. *Pinocchio Software Library.* `https://github.com/stack-of-tasks/pinocchio`.

[SDA18] Joan Sola, Jeremie Deray, and Dinesh Atchuthan. "A micro Lie theory for state estimation in robotics". In: *arXiv preprint arXiv:1812.01537* (2018).