## Master Thesis Project

## Ros-based control of a robotic leg for a quadruped robot

January 20, 2023
Author: Roberto Carta

Supervisor: Alba Perez Gracia
Date of the call: February 2023


ETSEIB
Escola Tècnica Superior
d’Enginyeria Industrial de Barcelona


## Review

The sector of Autonomous Mobile Robots (AMR) has grown a lot during the last years. In the literature an AMR is a robot able to move without any human operator control. With the improvements of the control systems, robots have gained a lot of dexterity and flexibility in the movements, migrating from restrictive mechanical systems like wheeling.
AMR with wheels are very efficient on plane grounds, like conventional industrial environments. Nevertheless, they lose efficiency when dealing with rough terrains like the ones you can find on mountain rescue, vineyards or building industry. A good alternative is to use legged robots, which imitate animal walking behaviour, for these types of terrain since they are able to easily overcome these obstacles.
The objective of this project is to create a control system for the robotic leg of a quadruped robot. A mechanical leg was developed and implemented at the CDEI for a quadruped robot, aimed for its locomotion in rugged and unknown terrain. This project will create the control system for this leg, so that it can execute the desired motions and it can be later integrated in the complete quadruped robot. The system will be designed so that it can be part of the stack of the quadruped robot. In this sense, the control systems software will be developed using the Robot Operating System (ROS) and MATLAB\&Simulink.

## Contents

1 Preface ..... 7
2 Introduction ..... 11
2.1 Objectives ..... 11
2.2 State of the art ..... 11
2.3 AMRs in primary industrial sector ..... 12
2.4 Description of legged robots ..... 13
2.5 Quadruped robots and existing commercial solutions ..... 14
2.6 Different types of gait ..... 16
2.6.1 Walk ..... 16
2.6.2 Trot ..... 17
2.6.3 Canter ..... 17
2.6.4 Gallop ..... 18
2.7 State of the art of control algorithms ..... 18
3 Mathematical model of the leg ..... 21
3.1 Description of the leg ..... 21
3.2 Direct Kinematics ..... 23
3.2.1 Denavit - Hartenberg Method ..... 23
3.3 Inverse Kinematics ..... 26
3.4 Differential Kinematics ..... 26
4 Kinematic Control ..... 29
4.1 Jacobian Inverse Algorithm ..... 29
4.2 Drawbacks and resolutions ..... 31
5 Experimental setup ..... 33
6 Software ..... 35
6.1 Robotic Operating System - ROS ..... 35
6.1.1 ROS vs ROS2 ..... 36
6.1.2 Communication between nodes and rqt_graph ..... 36
6.2 Gazebo Simulator ..... 37
6.3 Launch file ..... 37
6.4 Generating the URDF model from the 3D model ..... 38
6.5 Defining the controllers ..... 42
6.6 MATLAB\&Simulink ..... 44
6.6.1 Integration between Simulink and ROS ..... 45
6.7 Operational space trajectory generation ..... 47
7 Simulation results ..... 53
8 Experimental results ..... 59
9 Temporary planning and costs ..... 63
9.1 Cost of workers ..... 63
9.2 Energy Cost ..... 63
9.3 Software licenses cost ..... 64
9.4 Total cost of the project ..... 64
10 Environmental Impact ..... 65
11 Conclusions ..... 67
Bibliography ..... 69
A Appendix ..... 71
A. 1 URDF file of the robotic leg ..... 71
A. 2 MATLAB Code ..... 80
A. 3 C++ Code ..... 87
A. 4 ROS launch files ..... 89

## List of Figures

1 Guided robot (source: HepcoMotion) ..... 7
2 Mobile robot in a warehouse[Source: Robotics Business Review] ..... 8
3 Spot robot by Boston Dynamics ..... 8
4 Robotic leg developed by CDEI - UPC ..... 9
5 "Salto" monopod robot developed by Biomimetic Mill Systems Lab ..... 13
6 Tripod robot "Martian Petit Robot" developed by the University of Osaka ..... 13
7 Atlas Robot is one of the most famous biped robots (source Boston Dynamics) ..... 14
8 "Spot" quadruped robot by Boston Dynamics ..... 15
9 ANYmal robot during an Industrial inspection task (source ANYbotics) ..... 15
10 Cyberdog robot by Xiaomi[8] ..... 16
11 Walk pattern [9] ..... 17
12 Trot pattern [9] ..... 17
13 Canter pattern [9] ..... 17
14 Gallop pattern [9] ..... 18
15 3D model of the leg ..... 21
16 Schematic representation of the open chain kinematics of the robotic leg ..... 22
17 Denavit-Hartenberg kinematic parameters [15] ..... 23
18 Inverse kinematics algorithm with Jacobian inverse [15] ..... 30
19 Robotic leg developed by CDEI - UPC ..... 34
20 Ingenia EVE-XCR-E driver employed to control one motor ..... 34
21 Brushless motor employed in the leg to actuate a single joint ..... 34
22 ROS logo ..... 35
23 rqt_graph of this project ..... 37
24 3D model of the leg visualized in Solidworks ..... 38
25 Simplified 3D model without the parallel structure represented in Gazebo ..... 40
26 Tree representation of links connections in the simplified robotic leg description ..... 40
27 3D model of the leg including the transmission bar visualized in Gazebo ..... 41
28 Tree representation of the URDF description considering the parallel structure but not fully connected ..... 41
29 Simple Simulink implementation of the Jacobian Inverse algorithm ..... 45
30 Hierarchy representing the communication between MATLAB\&Simulink, ROS, Gazebo simulator and the real hardware ..... 46
31 Implementation of the Jacobian Inverse algorithm interfacing Simulink with ROS through routines of ROS-Toolbox ..... 47
32 Block scheme for publishing a topic in ROS network from Simulink ..... 47
33 Block scheme for subscribing to a topic in ROS network from Simulink ..... 47
34 rqt_graph of the whole control system ..... 47
35 Dog's foot trajectory with respect to camera reference frame ..... 48
36 Example of markers distribution on the body of a dog for motion capture high- lighting the relevant markers used in this project [Source: DeepAI] ..... 49
37 Generation of the source trajectory in Simulink to simulate the movement of a real dog's toe ..... 50
38 Dog foot trajectory represented with respect to the robot body reference frame ..... 51
39 Joint positions and velocities behaviour in Simulink simulation ..... 54
40 Error norm in cartesian space in Simulink simulation ..... 55
41 Comparison between desired trajectory $x_{d}$ and followed trajectory $x_{e}$ visualized in the $(x, z)$-plane ..... 55
42 Joint positions and velocities behaviour in Gazebo simulation ..... 56
43 Error norm in cartesian space in Gazebo simulation ..... 57
44 Comparison between desired trajectory $x_{d}$ and followed trajectory $x_{e}$ visualized in the $(x, z)$-plane, during Gazebo simulation ..... 57
45 Joint positions and velocities during experimental test ..... 60
46 Error norm in cartesian space during experimental test ..... 61
47 Comparison between desired trajectory $x_{d}$ and achieved trajectory $x_{e}$ visualized in the $(x, z)$-plane, during experimental simulation ..... 61

## 1 Preface

This project was developed inside the Centre de Disseny d'Equips Industrials (CDEI) of the Universitat Politecnica de Catalunya (UPC). The Center for Industrial Equipment Design, CDEIUPC is a technological innovation center of the Technical University of Catalonia, located in Barcelona. It has more than 21 years of experience carrying out research projects, technology transfer and training actions in the area of industrial equipment development and design as well as collaborating with companies from different sectors in the design, optimization, improvement and innovation of industrial equipment design. Throughout its history, the CDEI has had the opportunity to collaborate with small, medium and large companies, from different sectors such as industrial laundry, food sector, automotive, agriculture, renewable energy, public works machinery, metal coatings, packaging industry machinery, mobile robotics and indoor logistics, among many others. The Center has become an expert in connecting University knowledge and research with the needs and challenges of industry. On the research side, the Center is committed to research and innovation, especially in the area of applied mobile robotic solutions.

A robot is called Autonomous Mobile Robot (AMR) when it's capable of moving in an environment without the intervention of human control. There are several types of mobile robots, but we can essentially group them in 3 main categories depending on their type of locomotion:

1. Guided Robots
2. Wheeled Robots

## 3. Legged Robots

Guided robots are midway between industrial manipulators and AMRs. They are mainly constituted by a robotic manipulator which doesn't have a fixed base to the ground, but can move along mechanical guide, as reported in Figure 1. The main disadvantage is that they are not free to move in every directions but they are restricted to a predefined path.


Figure 1: Guided robot (source: HepcoMotion)
Wheeled robots are more dexterous so they can share their working space with other obstacles or humans in a safe way, like in Figure 2. Using special wheels, some robots can achieve omnidirectionality. These AMRs work well on simple terrains, like plane ones or small ramps. Nevertheless, they lose their efficiency in situations which present irregular terrains, such vine-
yards or alpine rescue．


Figure 2：Mobile robot in a warehouse［Source：Robotics Business Review］
Legged robots generally overcome the limitations introduced above as they can adapt to dif－ ferent terrains：like animals they can extend their legs in order to keep their body stable while walking on rough terrains．Moreover their legs can be raised in order to pass over obstacles or to jump over them．A simple example is when they have to walk up to a staircase，like in Figure 3．The throwback of this architecture is that it needs more actuators and the control is more difficult to perform．


Figure 3：Spot robot by Boston Dynamics

During the last years，CDEI realized several projects regarding wheeled AMRs，due to high demanding of this industrial sector．Following the development of many commercial legged robots it＇s worth to notice that many of them are limited to domestic use or inspection tasks． For this reason it＇s good to develop a robotic solution aimed to perform heavier tasks in the in－ dustrial and agricultural areas．The purpose of this whole research project，then，is to develop a competitive technology for customers that need that．
At the beginning of this work，the mechanical design and the production of a robotic leg has already been performed at CDEI，as shown in Figure 16，but without the implementation of any kind of control algorithm．The aim of this work is to continue the development of the leg on the software side，performing an high level control with the middleware ROS（Robot Operating System）．


Figure 4：Robotic leg developed by CDEI－UPC

## 2 Introduction

### 2.1 Objectives

At the beginning of this project, a mechanical leg has already been developed at CDEI, as well as the motor and relative drivers have already been implemented. Throughout this work a control system for this equipment will be designed, following several steps. The control algorithm will be designed considering that the leg will be implemented in a quadruped robot in the future. On a first phase an analysis of the literature will be made to check the state of the art of similar projects, including advanced control strategies and already existing commercial solutions. In this sense it is worth to develop a cartesian control of the foot of the leg.

The Robotic Operating System (ROS) will be the core middleware used in this work. The project will be divided in two major steps, namely simulation of the algorithm and implementation on the real mechanical system.

For this reason it is very useful to first develop a digital twin of the robotic leg and then use this model to perform some simulations in order to tune and correct the used algorithms. It will be necessary to export the already existing 3D model in an adequate format (URDF) which can be then uploaded in a robotic graphical simulator like Gazebo. For this reason, part of the project will be dedicated in building the accurate URDF model, to perform reliable tests.

Starting from the most naive objectives to the hardest ones, we will target to perform single joint movements (like position and velocity control) before applying higher level robotic algorithms. A kinematic control will be performed, for this reason it will be necessary to perform an analysis aimed at retrieving an accurate kinematic mathematical model which takes into account the leg parameters. To overcome the issue of coding the high level algorithm in a low level programming language, MATLAB and Simulink will be used to communicate with the ROS environment and perform the necessary control actions.

The implementation on the real mechanical system will be made with the aid of the low level drivers written by the company Beta Robots, which will help in focusing on high level control strategies. The final aim of the work will be to make the real system perform a trajectory similar to the one quadruped animals do when walking.

The development of walking algorithms which can make a quadruped robot adapt to harsh terrain won't be part of the project. Apart from the aim of the project, an environmental and social study, as well as a cost evaluation will be performed. A documentation will be also provided in order to make further improvements of the project easier.

### 2.2 State of the art

Like mentioned in the first section, improvements in control systems algorithms and the digitalization of factories in Industry 4.0 allowed autonomous mobile robots to gain some popularity in recent years. Nowadays, many companies have smart warehouses which exploit the flexibility of these new technologies to improve their logistic system. In the marine sector as well as in a warehouse environments the flow of materials is mainly performed in horizontal and plane directions, meaning that vertical movements are punctual or, in the majority of the cases, don't exist. All these circumstances made the use of wheeled AMRs one of the best solutions.

### 2.3 AMRs in primary industrial sector

In other areas, such as mining, agricultural and building industry as long as alpine rescue, it's possible to notice a growth in the digitalization and robotization of many processes even though with a lower impact compared to the secondary industrial sector. As this project can bring some improvements to the agricultural industry, it is worth to analyze this sector in this section. [1] The main use of these robots in agricultural applications is for collecting data relative to plants and vegetation (monitoring of weeds, sampling and health of the crops and of the fruits). All this information can help in improving efficiency of the farm activity, by reducing the use of resources and time waste due to inefficient organization. Recent progresses in computer vision (CV), machine learning (ML) and artificial intelligence (AI) techniques allow to process RGB images in order to get the information regarding the maturity of the fruits and their dimensions.

With the use of 3D data, especially given by LIDARs, it is possible to analyze the growth of trees and other slow growing crops. Knowledge of the spatial (3D) distribution of fruits through their detection and location, with different levels of resolution -within a specific tree and at plot levelis of enormous interest in agriculture. Having this information allows harvest and production estimates to be made, which leads to better planning of harvesting, storage and marketing tasks. [2]

By exploiting these technologies, among the others, it is possible to analyze the effects of different agricultural activities, get an estimate of the most adequate process for every area of the field or make useful predictions and estimations for scheduling fruit harvesting. All this system allows to optimize the resources, save money and reduce the environmental impact due to excessive use of pesticide and herbicide products.

Agriculture is currently undergoing a robotics revolution, but it is worth to notice that the majority of commercial AMRs nowadays uses wheeled locomotion systems. These robots suffer from known disadvantages: they are unable to move over rubble and steep or loose ground, and they trample continuous strips of land thereby reducing the viable crop area. [3]
For these reason legged robots seem better suited for these applications. To all this it must be added that the legged locomotion causes less crushing of the earth than wheels or caterpillars, thus avoiding the destruction of delicate crops such as native flora and pollinating fauna. [1]

Therefore, it can be concluded that, a robot with legs with sufficient autonomy to analyze large areas of land, optimized for agricultural tasks and with a suitable load capacity to transport sensors is a proposal that can offer added value to farmers and producers.

### 2.4 Description of legged robots

Robots with legs can be grouped according to the number of legs used. In principle a robot with legs can have any arbitrary number of limbs, although generally, (probably for inspiration from nature) designers usually choose even numbers. Monopod and tripods robots are not so common but exist anyway (See Figures 5 and 6).


Figure 5: "Salto" monopod robot developed by Biomimetic Mill Systems Lab


Figure 6: Tripod robot "Martian Petit Robot" developed by the University of Osaka
The most common legged robots are bipeds or quadrupeds. Generally, the common challenges of biped robots include, but are not limited to, the following:

- Unstable structures due to the passive joints located at the unilateral contact between the foot and the ground
- Need of more complex control algorithm in order to balance their unstable structure
- Combination and transition between the different modes of bipedal locomotion: walking

ETSEIB
and running without falling

- Bipedal robots have multiple degrees of freedom (DOFs). Most researchers use simplified models to reach a trade-off between simplicity and the dexterity [4]

It is also very common for these types of robot to be equipped of large counterweights, gyroscopes or other systems to facilitate control of the whole.


Figure 7: Atlas Robot is one of the most famous biped robots (source Boston Dynamics)
On the other hand, robots with three or more legs can rest in a stable position even with the lack of control and in some circumstances with low consumption by the actuators.
Quadruped walking robots provide better stability both while walking and resting, and payload capacity over other legged robots. These kinds of robots can be used to travel in rocky, muddy and sloped terrains.[5]

Hexapod robots require more independent actuators adding complexity to both the design and to the control. In addition, a greater number of limbs also leads to an increase in the mass of the robot. This alternative is suitable for robots where speed is not a priority and they must guarantee a very stable gait and/or a lot of grip with the ground.

### 2.5 Quadruped robots and existing commercial solutions

The best-known example of a robot with legs is the quadruped "Spot" developed by Boston Dynamics. This robot is capable of moving at speeds of $1.6 \mathrm{~m} / \mathrm{s}$ and has a wide variety of accessories for inspection or even a manipulator arm. It is capable of handling up to 14 kg payloads. This robot has proven to be functional in a large number of situations, like alpine rescue, space
missions and exploration of other risky environments due to its bioinspired dynamic control. It is very suitable also for inspecting dusty environments due to its IP54 degree of protection. It can be remotely controlled by a human operator while also being able to navigate and perform some tasks autonomously since it's equipped with a 3D vision system with SLAM and obstacle avoidance algorithms. Due to its Omni-directional walking and multiple walking and trotting gaits it can climb and descend stairs.
One of the main disadvantages of this robot is the price ( $\$ 74.500$ in its simplest version without add-ons). The maximum autonomy of a battery is 90 minutes of travel, which is why it is important that the batteries are easily removable to be able to use spare parts. [6]


Figure 8: "Spot" quadruped robot by Boston Dynamics
ANYmal is a quadrupedal robot designed by the company ANYbotics for autonomous operation in challenging environments. . Thanks to incorporated laser sensors and cameras, the robot can perceive its environment to continuously create maps and accurately localize. Based on this information, it can autonomously plan its navigation path and carefully select footholds while walking. It was first developed for industrial inspection of oil and gas sites. It carries batteries for more than 2 h autonomy and different sensory equipment such as optical and thermal cameras, microphones, gas-detection sensors and active lighting. With this payload, the machine weighs less than 30 kg and can hence be easily transported and deployed by a single operator. [7]


Figure 9: ANYmal robot during an Industrial inspection task (source ANYbotics)

Other companies have also developed small-sized quadruped robots that mimic the movement of the previously mentioned robots but at a much lower price. An example is the "Cyberdog" (Figure 10) presented in August 2021 by the company Xiaomi. This 3 kg robot has several builtin cameras and can move at speeds slightly above $3 \mathrm{~m} / \mathrm{s}$. Its selling price is slightly higher than $1,200 €$. It should be noted that this model has no load-carrying capacity and the incorporated sensors are limited to image and sound. This, along with the very limited production series, makes its field applications few and it is considered that this product is a promotion campaign for future models with more performance.[8]


Figure 10: Cyberdog robot by Xiaomi[8]

### 2.6 Different types of gait

Not all creatures move the same way; some gaits are better adapted to keeping up over longer distance, others for explosive movement and sprinting. Clearly, Mother Nature had two main concerns in evolving solid gait styles: how fast can we move and how energy-efficient is that movement. Especially for predators, if they expend more energy than they stand to gain by reaching the food, moving wouldn't be worth it.
Quadrupeds can have different variations of gait based on a few variables: speed, personal preference, and training. The analysis below of each stride will include a brief description of the gait with some important notes; the footfall analysis of the gait (the sequence in which feet touch and lift from the ground); a representation of the footfall and a reference of the gait. Each type of gait can evolve to another faster or slower gait. Some types of faster gaits are more favorable to transition into, depending on which gait the animal is coming from and the similarity of the footfall pattern. [9] In quadrupeds we can distinguish basically between 4 types of gaits: walk, trot, gallop, canter. [10]

### 2.6.1 Walk

The walk is the slowest and most energy-efficient gait. While walking, the animal always has two to three points of contact with the ground. During the walk, each hind foot is alternately replacing the front foot on the same side, switching, and repeating the same action on the opposite side. During the switch, the quadruped will necessarily have only two points of contact with the ground for a split second. This is also the most stable gait, due to the fact that the quadruped can always have three feet on the ground. [9]


Figure 11: Walk pattern [9]

### 2.6.2 Trot

With the trot, the animal begins to gain some speed, although still not committed to running, and maintains an energy-efficient approach. We can compare it to the human version of jog-ging-midway between walking and sprinting. Much like jogging, it's the preferred way to cover longer distances in a relatively short time, in contrast to the gallop in which a lot of energy is consumed in short bursts before being forced to stop. In a trained horse, a trot can be sustained potentially for hours. The trot is a two-beat gait that follows a diagonal pattern, with the limbs opposing diagonally hitting the ground simultaneously and then alternating. [9]


Figure 12: Trot pattern [9]

### 2.6.3 Canter

The canter is a three-beat gait that falls between the trot and the gallop in terms of speed, and this is also reflected in the sequence of footfalls. With three beats on each stride, the canter is an asymmetrical gait, which means that each animal can have either a left or a right lead, depending on which front limb is leaving the ground last on each stride. In the canter, we find the rhythm of beats as one-two-one, followed by a suspension where all feet have left the ground. As the speed increases, the period of suspension is longer until the animal reaches a speed where it is forced to switch to a full gallop. [9]


Figure 13: Canter pattern [9]

### 2.6.4 Gallop

The gallop is the fastest gait; each stride can cover more ground than in other gaits, but it's also very energy demanding and less useful for long-distance traveling. The footfall is an evolution of the canter and its natural progression, except that the gallop becomes a four-beat gait. Each footfall becomes equidistant from the others and is always followed by a moment of suspension at the end. The two back legs hit the ground first with slightly different timing while the two front follow in the same pattern immediately after. [9]


Figure 14: Gallop pattern [9]

### 2.7 State of the art of control algorithms

The control of a robotic leg poses several challenges, such as stability, accuracy, efficiency, and robustness to uncertainty and disturbances. To address these challenges, various control algorithms have been proposed and applied in the literature. In this chapter, we review the most popular and effective control algorithms for robotic legs, and discuss their advantages and limitations. Based on a model prediction, the Model Predictive Control(MPC) framework easily incorporates system dynamics and constraints by transcribing the control law as a constrained optimization problem. Recent applications of MPC on quadrupeds have shown the capability of MPC in planning and controlling complex dynamic motions while embracing system dynamics and constraints arising from friction and motor saturation.[11] However, these controllers assume accurate knowledge of the dynamic model, or in other words, do not address substantial model uncertainty in the system. Many safetycritical missions, such as firefighting, disaster response, exploration, require the robot to operate swiftly and stably while dealing with high levels of uncertainty and large external disturbances.Adaptive control can address model uncertainty in control systems. [12] Impedance control is a control strategy that regulates the interaction between the robot and the environment. It adjusts the impedance of the robot to match the impedance of the environment and reduce the external forces acting on the leg. The objective of impedance control is not to directly control position or force, but the relationship between them. This allows reducing or increasing apparent stiffness, damping, or mass depending on the task. Impedance control has been applied to various types of robotic legs involved into bipedal robots, quadruped robots, and exoskeletons. [13] Reinforcement learning (RL) is a type of machine learning technique that allows robots to learn from their own experience and optimize their control strategies. The algorithm utilizes a reward function, which encodes the desired performance of the robot, to adjust its control policy through trial and error. As the robot learns and adapts to new experiences, it maximizes the reward function and improves its overall control performance. This technique has been applied to control robotic legs with the goal of making them more autonomous and efficient. If successfully applied, RL can automate the controller design, completely removing the need for system identification, and resulting in gaits that are directly optimized for a particular robot and environment. However, applying RL to learning gaits in the real world is challenging, since current algorithms often require a large

ETSEIB
number of samples-on the order of tens of thousands of trials. Moreover, such algorithms are often highly sensitive to hyperparameter settings and require considerable tuning. [14] The research on RL for controlling robotic legs is still ongoing and new advances are being made regularly.

In conclusion, there are several state-of-the-art algorithms that are commonly used to control a robotic leg, each with its own advantages and limitations. MPC can handle constraints and take into account the dynamics of the leg, adaptive control is useful for uncertain or changing environments, impedance control is efficient in regulating the interaction between the robot and the environment, and reinforcement learning is useful to optimize robot trajectory generation without needing a mathematical model. However, the choice of control algorithm depends on the specific requirements of the robotic leg and the task it is meant to perform.

## 3 Mathematical model of the leg

### 3.1 Description of the leg

The robotic leg, as depicted in Figure 15, consists of six links, three revolute active joints and three passive joints. The mechanical design presents a parallel structure that has been integrated in order to regulate the movement of the joint between Link 2 and Link3B. This structure utilizes a motor that is situated in proximity to the shoulder of the leg, with the intention of optimizing the effort required by the initial two motors in order to move the first two joints. The parallel structure in question connects the passive joint between Link2 and Link3A with the joint between Link2 and Link4, enabling the manipulation of the active joint to control the movement of the passive joint. This design decision was made to improve the efficiency and performance of the robotic leg, by optimizing the effort required by the motors and allowing for a more compact design. From a mathematical and analytical standpoint, the parallel structure can be described as a planar four-bar parallelogram. This classification leads to the conclusion that the passive joint exhibits similar behavior to that of the controlled joint, as the movement of one joint can be replicated through manipulation of the other. As a result, the kinematic analysis of the robotic leg can be conducted using the simplified model of the leg reported in Figure 16.
It should be noticed that Link4 has its own moment of inertia, meaning the effort applied by the motor is not the same as that applied at the passive joint in terms of dynamics. The kinematics parameters of the links are reported in Table 1. Due to the mechanical structure of the leg it has to be considered that there exist some physical limits on the joint positions: with reference to the structure of the system shown in Figure 16, the joints operational ranges are reported in Table 2.


Figure 15: 3D model of the leg


Figure 16: Schematic representation of the open chain kinematics of the robotic leg

| Parameter | Value |
| :---: | :---: |
| $l_{1, B}$ | 100 mm |
| $l_{1, A}$ | $78,75 \mathrm{~mm}$ |
| $l_{2}$ | 300 mm |
| $l_{3}$ | 300 mm |

Table 1: Kinematic parameters of the leg

| Joint | Range[rad] |
| :---: | :---: |
| $\theta_{1}$ | $[-\pi, 0,30]$ |
| $\theta_{2}$ | $[-\pi, 1,00]$ |
| $\theta_{3}$ | $[0,67,2,27]$ |

Table 2: Operational limits of the joints

### 3.2 Direct Kinematics

In the following section, the kinematic equations of the leg will be derived. The objective of this analysis is to establish the mathematical representation of the location of the toe of the leg at point $\boldsymbol{P}$ with respect to the reference frame fixed to the body of the quadruped $\left(R F_{R}\right)$. This reference frame, represented in Figure 16 by the coordinates ( $x_{R}, y_{R}, z_{R}$ ), is located at the shoulder of the leg and it has the same origin of the $R F_{0}$. The kinematic equations will describe the relationship between the position of the toe and the vector of generalized coordinates ( $q_{1}, q_{2}, q_{3}$ ) that define the configuration of the leg. It is worth to notice that, since only revolute joints are present in this mechanical structure then $\left(q_{1}, q_{2}, q_{3}\right)=\left(\theta_{1}, \theta_{2}, \theta_{3}\right)$. Also the equations which determine the inverse kinematic will be presented, in order to find which values of $\left(q_{1}, q_{2}, q_{3}\right)$ in the joint space are necessary to reach a certain point in the operational space. It must be noticed that the point $\boldsymbol{P}$ is not the real contact point of the leg with the ground, instead it models the center of a sphere with radius $r$, as the case of the gum ball in our leg. On ideal terrains, the contact point can be modelled easily but this not the case for arbitrary type of terrains, indeed the contact with the ground can deform the toe made of gum and this will depend on many factors, such as robot's payload and stiffness of the ground, among the others.

### 3.2.1 Denavit - Hartenberg Method

To express the kinematic equations of a chained robotic structure it is possible to use the DenavitHartenberg method. This allows to represent the pose of the end-effector of a manipulator with respect to a fixed reference frame through a recursive algorithm. Indeed it is possible to find the kinematic relationship between 2 consecutive links describing their relative roto-translation and represent it with a homogeneous transformation matrix:

$$
\left[\begin{array}{cc}
\boldsymbol{R} & \boldsymbol{T}  \tag{1}\\
\mathbf{0}^{T} & 1
\end{array}\right] \in S E(3)
$$



Figure 17: Denavit-Hartenberg kinematic parameters [15]
With reference to Figure 17, let Axis $i$ denote the axis of the joint connecting Link $i-1$ to Link $i$; the so-called Denavit-Hartenberg convention (DH) is adopted to define link Frame $i$ :

- Choose axis $z_{i}$ along the axis of Joint $i+1$
- Locate the origin $O_{i}$ at the intersection of axis $z_{i}$ with the common normal to axes $z_{i-1}$ and $z_{i}$. Also, locate $O_{i^{\prime}}$ at the intersection of the common normal with axis $z_{i-1}$.
- Choose axis $x_{i}$ along the common normal to axes $z_{i-1}$ and $z_{i}$ with positive direction from Joint $i$ to Joint $i+1$.
- The $y_{i}$ axis is given by the outer product $z_{i} \otimes x_{i}$ in order to get a right-handed frame. [15]

With this procedure, to make a coordinate transformation from the first reference frame $i-1$ to the reference frame $i$, two roto-translations are performed. First a roto-translation relative to the $z_{i-1}$ axis is made. This is described by the following homogeneous transformation matrix:

$$
\boldsymbol{A}_{i^{\prime}}^{i-1}=\left[\begin{array}{cccc}
\cos \theta_{i} & -\sin \theta_{i} & 0 & 0  \tag{2}\\
\sin \theta_{i} & \cos \theta_{i} & 0 & 0 \\
0 & 0 & 1 & d_{i} \\
0 & 0 & 0 & 1
\end{array}\right]
$$

Consequently a roto-translation relative to the axis $x_{i^{\prime}}$ is performed, which is represented by the following homogeneous transformation matrix:

$$
\boldsymbol{A}_{i}^{i^{\prime}}=\left[\begin{array}{cccc}
1 & 0 & 0 & a_{i}  \tag{3}\\
0 & \cos \alpha_{i} & -\sin \alpha_{i} & 0 \\
0 & \sin \alpha_{i} & \cos \alpha_{i} & 0 \\
0 & 0 & 0 & 1
\end{array}\right]
$$

The resulting coordinate transformation is obtained by postmultiplication of the single transformations as:

$$
\boldsymbol{A}_{i}^{i-1}\left(q_{i}\right)=\boldsymbol{A}_{i^{\prime}}^{i-1} \boldsymbol{A}_{i}^{i^{\prime}}=\left[\begin{array}{cccc}
\cos \theta_{i} & -\sin \theta_{i} \cos \alpha_{i} & \sin \theta_{i} \sin \alpha_{i} & a_{i} \cos \theta_{i}  \tag{4}\\
\sin \theta_{i} & \cos \theta_{i} \cos \alpha_{i} & -\cos \theta_{i} \sin \alpha_{i} & a_{i} \sin \theta_{i} \\
0 & \sin \alpha_{i} & \cos \alpha_{i} & d_{i} \\
0 & 0 & 0 & 1
\end{array}\right]
$$

Adopting this notation, the transformation matrices between consecutive reference frames in this robotic leg are:

$$
\begin{gather*}
\boldsymbol{A}_{1}^{0}=\left[\begin{array}{cccc}
-\sin \left(q_{1}\right) & 0 & \cos \left(q_{1}\right) & 0 \\
\cos \left(q_{1}\right) & 0 & \sin \left(q_{1}\right) & 0 \\
0 & 1 & 0 & l_{1, B} \\
0 & 0 & 0 & 1
\end{array}\right]  \tag{5}\\
\boldsymbol{A}_{2}^{1}=\left[\begin{array}{cccc}
\cos \left(q_{2}\right) & \sin \left(q_{2}\right) & 0 & l_{2} \cos \left(q_{2}\right) \\
\sin \left(q_{2}\right) & -\cos \left(q_{2}\right) & 0 & l_{2} \sin \left(q_{2}\right) \\
0 & 0 & -1 & l_{1, A} \\
0 & 0 & 0 & 1
\end{array}\right] \tag{6}
\end{gather*}
$$

ETSEIB

$$
\boldsymbol{A}_{3}^{2}=\left[\begin{array}{cccc}
\cos \left(q_{3}\right) & -\sin \left(q_{3}\right) & 0 & l_{3} \cos \left(q_{3}\right)  \tag{7}\\
\sin \left(q_{3}\right) & \cos \left(q_{3}\right) & 0 & l_{3} \sin \left(q_{3}\right) \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{array}\right]
$$

Finally, taking into account that the origin of the robot Reference Frame $\left(R F_{R}\right)$ will be coincident to the origin of $R F_{0}$ and it's oriented with Axis $x_{R}$ representing the robot progressing direction and the Axis $z_{R}$ normal to the ground, we can introduce an additional static homogeneous transformation matrix:

$$
\boldsymbol{A}_{0}^{R}=\left[\begin{array}{cccc}
0 & 0 & -1 & 0  \tag{8}\\
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{array}\right]
$$

Referring to $\sin (\cdot)$ as $s(\cdot)$ and to $\cos (\cdot)$ as $c(\cdot)$, the final transformation matrix will be given by:

$$
\begin{gather*}
\boldsymbol{A}_{3}^{R}=\boldsymbol{A}_{0}^{R} \boldsymbol{A}_{1}^{0} \boldsymbol{A}_{2}^{1} \boldsymbol{A}_{3}^{2}= \\
{\left[\begin{array}{cccc}
-s\left(q_{2}-q_{3}\right) & c\left(q_{2}-q_{3}\right) & 0 & -l_{1, B}-l_{3} s\left(q_{2}-q_{3}\right)-l_{2} s\left(q_{2}\right) \\
-c\left(q_{2}-q_{3}\right) s\left(q_{1}\right) & -s\left(q_{2}-q_{3}\right) s\left(q_{1}\right) & -c\left(q_{1}\right) & -l_{3} s\left(q_{1}\right) c\left(q_{2}-q_{3}\right)-l_{2} s\left(q_{1}\right) c\left(q_{2}\right)+l_{1, A} c\left(q_{1}\right) \\
-c\left(q_{2}-q_{3}\right) c\left(q_{1}\right) & -s\left(q_{2}-q_{3}\right) c\left(q_{1}\right) & s\left(q_{1}\right) & -l_{3} c\left(q_{1}\right) c\left(q_{2}-q_{3}\right)-l_{2} c\left(q_{1}\right) c\left(q_{2}\right)-l_{1, A} s\left(q_{1}\right) \\
0 & 0 & 0 & 1
\end{array}\right]} \tag{9}
\end{gather*}
$$

With these equations it is possible to express the position of the point $\boldsymbol{P}$ with respect to the Reference Frame of the robot, taking into account that $\boldsymbol{P}$ is positioned at the origin of the Reference Frame 3. In homogeneous coordinates we have:

$$
\boldsymbol{P}^{3}=\left[\begin{array}{l}
0  \tag{10}\\
0 \\
0 \\
1
\end{array}\right]
$$

and consequently:

$$
\boldsymbol{P}^{R}=\boldsymbol{A}_{3}^{R} \boldsymbol{P}^{3}=\left[\begin{array}{c}
P_{x}  \tag{11}\\
P_{y} \\
P_{z} \\
1
\end{array}\right]=\left[\begin{array}{c}
-l_{1, B}-l_{3} s\left(q_{2}-q_{3}\right)-l_{2} s\left(q_{2}\right) \\
-l_{3} s\left(q_{1}\right) c\left(q_{2}-q_{3}\right)-l_{2} s\left(q_{1}\right) c\left(q_{2}\right)+l_{1, A} c\left(q_{1}\right) \\
-l_{3} c\left(q_{1}\right) c\left(q_{2}-q_{3}\right)-l_{2} c\left(q_{1}\right) c\left(q_{2}\right)-l_{1, A} s\left(q_{1}\right) \\
1
\end{array}\right]
$$

Formula 11 expresses the equations which rule the point's position forwad kinematics of the leg. It is worth to notice that this defines a non-linear map $\boldsymbol{x}_{e}=\boldsymbol{k}(\boldsymbol{q})$ and relates the joint variables $\left(\boldsymbol{q}_{1}, \boldsymbol{q}_{2}, \boldsymbol{q}_{3}\right)$ to $\boldsymbol{x}_{e}$. As in this case we are describing only the end-effector position, then $\boldsymbol{x}_{e}=\boldsymbol{p}_{e}$.

### 3.3 Inverse Kinematics

Starting from the direct kinematics relationship in 11 it is possible to retrieve the vector of joint variables $\left(q_{1}, q_{2}, q_{3}\right)$ which allow to position the robot end-effector in a predefined position in the operational space $\boldsymbol{x}_{e}$.

$$
\begin{gather*}
q_{1}=\arctan \left(\frac{-P_{z}}{P_{y}}\right) \pm \arccos \left(\frac{l_{1, A}}{\sqrt{P_{y}^{2}+P_{z}^{2}}}\right)  \tag{12}\\
q_{2}=\arctan \left(\frac{2 l_{2}\left(l_{1, B}+P_{x}\right)}{2\left(P_{z} c\left(q_{1}\right)+P_{y} s\left(q_{1}\right) l_{2}\right.}\right) \pm \arccos \left(\frac{l_{3}^{2}-l_{2}^{2}-\left(l_{1, B}+P x\right)^{2}-\left(P_{z} c\left(q_{1}\right)+P_{y} s\left(q_{1}\right)^{2}\right.}{\sqrt{\left(2 l_{2}\left(l_{1, B}+P_{x}\right)\right)^{2}+\left(2\left(P_{z} c\left(q_{1}\right)+P_{y} s\left(q_{1}\right) l_{2}\right)^{2}\right.}}\right)  \tag{13}\\
q_{3}=\arctan \left(\frac{l_{2} s\left(q_{2}\right)+l_{1, B}+P_{x}}{l_{2} c\left(q_{2}\right)+P_{z} c(q 1)+P_{y} s\left(q_{1}\right)}\right)-q_{2} \tag{14}
\end{gather*}
$$

It is important to note that for any desired operational space position $\left(\boldsymbol{P}_{\boldsymbol{x}}, \boldsymbol{P}_{\boldsymbol{y}}, \boldsymbol{P}_{\boldsymbol{z}}\right)^{T}$, there exist four possible sets of joint angles $\left(\boldsymbol{q}_{1}, \boldsymbol{q}_{2}, \boldsymbol{q}_{3}\right)^{T}$ that would result in the end-effector reaching that position. To ensure that the robotic leg follows the desired cartesian trajectory, it is crucial to provide the correct initial condition for the inverse kinematics problem. This can be accomplished through the use of constraints, and by properly initializing the inverse kinematics algorithm. As long as the system does not pass through a singularity, the correct solution among the multiple possible solutions can be determined.

### 3.4 Differential Kinematics

The goal of the differential kinematics is to find the relationship between the joint velocities and the end-effector linear and angular velocities. In other words, it is desired to express the end-effector linear velocity $\dot{\boldsymbol{p}}_{e}$ and angular velocity $\omega_{e}$ as a function of the joint velocities $\dot{\boldsymbol{q}}$. The sought relations are both linear in the joint velocities, i.e.,

$$
\begin{align*}
& \dot{p}_{e}=J_{P}(q) \dot{q}  \tag{15}\\
& \omega_{e}=J_{O}(q) \dot{q} \tag{16}
\end{align*}
$$

In (15) $J_{P}$ is the $(3 \times n)$ matrix relating the contribution of the joint velocities $\dot{\boldsymbol{q}}$ to the endeffector linear velocity $\dot{\boldsymbol{p}}_{e}$, while in (16) $J_{O}$ is the $(3 \times n)$ matrix relating the contribution of the joint velocities $\dot{\boldsymbol{q}}$ to the end-effector angular velocity $\omega_{e}$. In compact form, 15,16 can be written as

$$
v_{e}=\left[\begin{array}{c}
\dot{p}_{e}  \tag{17}\\
\omega_{e}
\end{array}\right]=J(q) \dot{q}
$$

which represents the manipulator differential kinematics equation. The ( $6 \times n$ ) matrix $\boldsymbol{J}$ is the manipulator geometric Jacobian. The analytical Jacobian instead is obtained as:

$$
\begin{equation*}
J_{A}(q)=\frac{\partial k(q)}{\partial q} \tag{18}
\end{equation*}
$$

In general the geometric and the analytical Jacobian are different since the following hold:

$$
\begin{align*}
& \dot{p}_{e}=\frac{d p_{e}}{d t}  \tag{19}\\
& \omega_{e} \neq \frac{d \phi_{e}}{d t} \tag{20}
\end{align*}
$$

but in this project we are not describing the orientation of the end-effector $\phi_{e}$ so the two Jacobians are the same and they can be retrieved by time differentiation of $\boldsymbol{k}(\boldsymbol{q})$ :

$$
\begin{gather*}
\boldsymbol{J}(\boldsymbol{q})=\boldsymbol{J}_{\boldsymbol{A}}(\boldsymbol{q})=\frac{\boldsymbol{\partial k}(\boldsymbol{q})}{\boldsymbol{\partial \boldsymbol { q }}}= \\
{\left[\begin{array}{ccc}
0 & -l_{2} c\left(q_{2}\right)-l_{3} c\left(q_{2}-q_{3}\right) & l_{3} c\left(q_{2}-q_{3}\right) \\
-l_{1, A} s\left(q_{1}\right)-l_{2} c\left(q_{1}\right) c\left(q_{2}\right)-l_{3} c\left(q_{1}\right) c\left(q_{2}-q_{3}\right) & s\left(q_{1}\right)\left(l_{2} s\left(q_{2}\right)+l_{3} s\left(q_{2}-q_{3}\right)\right) & -l_{3} s\left(q_{2}-q_{3}\right) s\left(q_{1}\right) \\
l_{2} c\left(q_{2}\right) s\left(q_{1}\right)-l_{1, A} c\left(q_{1}\right)+l_{3} s\left(q_{1}\right) c\left(q_{2}-q_{3}\right) & c\left(q_{1}\right)\left(l_{2} s\left(q_{2}\right)+l_{3} s\left(q_{2}-q_{3}\right)\right) & -l_{3} s\left(q_{2}-q_{3}\right) c\left(q_{1}\right)
\end{array}\right]} \tag{21}
\end{gather*}
$$

The identification and analysis of Jacobian singularities will be presented in Section 4.2, along with the methods and techniques used to prevent the system from falling into such singular configurations. The singularities in Jacobian matrix can cause the loss of control of the robotic arm and lead to unexpected behaviors, thus it is important to have a proper understanding of these singularities and the methods to avoid them in order to achieve stable and accurate control of the robotic leg.

## 4 Kinematic Control

The differential kinematics equation represents a linear mapping between the joint velocity space and the operational velocity space, although it varies with the current configuration. This fact suggests the possibility to utilize the differential kinematics equation to tackle the inverse kinematics problem. [15] By considering equation (17) with $J(q)$ square as in our case, the joint velocities can be obtained via simple inversion of the Jacobian matrix:

$$
\begin{equation*}
\dot{q}=J^{-1}(q) v_{e} \tag{22}
\end{equation*}
$$

If the initial manipulator posture $\boldsymbol{q}(\mathbf{0})$ is known, joints positions can be computed by integrating velocities over time, i.e.,

$$
\begin{equation*}
q(t)=\int_{0}^{t} \dot{q}(\sigma) d \sigma+q(0) \tag{23}
\end{equation*}
$$

The integration can be performed in discrete time by resorting to numerical techniques. The simplest technique is based on the Euler integration method; given an integration interval $\Delta t$, if the joint positions and velocities at time $t_{k}$ are known, the joint positions at time $t_{k+1}=t_{k}+\Delta t$ can be computed as

$$
\begin{equation*}
q\left(t_{k+1}\right)=q\left(t_{k}\right)+\dot{q}\left(t_{k}\right) \Delta t \tag{24}
\end{equation*}
$$

and so recalling 22 we have

$$
\begin{equation*}
q\left(t_{k+1}\right)=q\left(t_{k}\right)+J^{-1}\left(q\left(t_{k}\right)\right) v_{e}\left(t_{k}\right) \Delta t \tag{25}
\end{equation*}
$$

It follows that the computed joint velocities $\dot{\boldsymbol{q}}$ do not coincide with those satisfying 22 in the continuous time. Therefore, reconstruction of joint variables $\boldsymbol{q}$ is entrusted to a numerical integration which involves drift phenomena of the solution; as a consequence, the end-effector pose corresponding to the computed joint variables differs from the desired one. For this reason it's necessary to introduce a control scheme that accounts for the operational space error between the desired and the actual end-effector position.

### 4.1 Jacobian Inverse Algorithm

Let the operational space error between the desired and the actual end-effector position be:

$$
\begin{equation*}
e=x_{d}-x_{e} \tag{26}
\end{equation*}
$$

Consider the time derivative of such error:

$$
\begin{equation*}
\dot{e}=\dot{x}_{d}-\dot{x}_{e} \tag{27}
\end{equation*}
$$

which, according to the differential kinematics can be written as:

$$
\begin{equation*}
\dot{e}=\dot{x}_{d}-J_{A}(q) \dot{q} \tag{28}
\end{equation*}
$$

On the assumption that matrix $\boldsymbol{J}_{\boldsymbol{A}}$ is square and nonsingular, the choice

$$
\begin{equation*}
\dot{q}=J_{A}^{-1}(q)\left(\dot{x}_{d}+K e\right) \tag{29}
\end{equation*}
$$

leads to the equivalent linear system

$$
\begin{equation*}
\dot{e}+K e=0 \tag{30}
\end{equation*}
$$

If $\boldsymbol{K}$ is a positive definite (usually diagonal) matrix, the system (30) is asymptotically stable. The error tends to zero along the trajectory with a convergence rate that depends on the eigenvalues of matrix $\boldsymbol{K}$; the larger the eigenvalues, the faster the convergence.

The matrix $\boldsymbol{K}$ in this project was chosen to be:

$$
K=\left[\begin{array}{ccc}
0.8 & 0 & 0  \tag{31}\\
0 & 0.8 & 0 \\
0 & 0 & 0.8
\end{array}\right]
$$

which ensures the convergence to 0 of the error and at the same time keeps the joint velocities $\dot{\boldsymbol{q}}$ limited.


Figure 18: Inverse kinematics algorithm with Jacobian inverse [15]
The block diagram corresponding to the inverse kinematics algorithm in equation (29) is shown in Figure 18, where $\boldsymbol{k}(\cdot)$ represents the direct kinematics function. This diagram can be analyzed in terms of standard feedback control schemes. Specifically, it can be seen that the nonlinear block $\boldsymbol{k}(\cdot)$ is required to calculate $\boldsymbol{x}_{e}$ and therefore the tracking error $\boldsymbol{e}$, while the block $J_{\boldsymbol{A}}^{-1}(q)$ has been included to compensate for $\boldsymbol{J}_{\boldsymbol{A}}(\boldsymbol{q})$ and make the system linear. The block diagram illustrates the presence of a series of integrators in the forward loop, which ensures that the steady-state error is zero when the reference $\left(\dot{x}_{\boldsymbol{d}}=0\right)$ is constant. Additionally, the feedforward action provided by $\dot{\boldsymbol{x}}_{\boldsymbol{d}}$ for a time-varying reference ensures that the error remains zero (in the case $(e(0)=0)$ throughout the entire trajectory, regardless of the type of desired reference $x_{d}(t)$.

The scheme in Figure 18 can implement a kinematic control, provided that the integrator is regarded as a simplified model of the robot, thanks to the presence of single joint local servos, which ensure a more or less accurate reproduction of the velocity commands. It is important to note that the proposed kinematic control technique is suitable for achieving satisfactory performance when the required motion is not too fast or the required accelerations are not too high. This is because, under these conditions, the dynamics of the system can be ignored as the accelerations are relatively insignificant. However, when high-speed motions or rapid accelerations are required, it is necessary to take the dynamics of the system into consideration and use a more advanced control technique, such as a dynamic control technique, to achieve stable and accurate control of the robotic leg.

### 4.2 Drawbacks and resolutions

Both solutions (22) and (29) can be computed only when the Jacobian has full rank. Hence, they become meaningless when the manipulator is at a singular configuration; in such a case, the system $\boldsymbol{v}_{\boldsymbol{e}}=\boldsymbol{J} \dot{\boldsymbol{q}}$ contains linearly dependent equations. It is crucial to note that the inversion of the Jacobian may not only be problematic at a singularity, but also in the neighborhood of a singularity. For instance, the computation of the Jacobian inverse requires the calculation of the determinant, which may become relatively small in the neighborhood of a singularity, leading to potentially large joint velocities.

In our case the singularities are present in the following configurations when the Jacobian is rank deficient:

$$
\begin{align*}
& q_{3}=0  \tag{32}\\
& q_{3}=\pi \tag{33}
\end{align*}
$$

This means that when the Link3A and the Link2 are alligned we can be in a singular configuration. This practically can never happen due to physical limits in the joint movement but anyway it is better to avoid configurations near to the singular ones. For this reason, it was decided to start the robotic leg in the joint configuration $\left(q_{1}, q_{2}, q_{3}\right)=\left(0,0, \frac{\pi}{2}\right)$.

## 5 Experimental setup

The robotic leg, which has already been developed and is currently hosted at CDEI, is depicted in Figure 19. It is equipped with three brushless motors, each of which is outfitted with a harmonic drive to facilitate movement of the leg's three joints. Each motor is equipped with an incremental encoder, which is mounted on the motor side, and is actuated by an electronic driver that utilizes the EtherCAT communication protocol to communicate with the associated desktop PC. In Tables 5, 6, 7 some useful data regarding the motors, their gearboxes and the electronic drivers are reported. The electricity source comes from a power supply with 48 V output voltage and 30 A maximum current. It is worth to notice that, as already mentioned in Section 3.1 each active joint in the leg presents physical limits due to the mechanical structure of the system. These limits are reported in Table 4. Moreover the kinematics parameters for every link of the leg are summarized in Table 3.

| Parameter | Value |
| :---: | :---: |
| $l_{1, B}$ | 100 mm |
| $l_{1, A}$ | $78,75 \mathrm{~mm}$ |
| $l_{2}$ | 300 mm |
| $l_{3}$ | 300 mm |

Table 3: Kinematic parameters of the leg

| Joint | Range $[\mathrm{rad}]$ |
| :---: | :---: |
| $\theta_{1}$ | $[-\pi, 0,30]$ |
| $\theta_{2}$ | $[-\pi, 1,00]$ |
| $\theta_{3}$ | $[0,67,2,27]$ |

Table 4: Operational limits of the joints

| Producer | Mavilor Motors |
| :---: | :---: |
| Model type | BR-02 |
| Mass $[\mathrm{kg}]$ | 0,47 |
| Maximum motor speed $[\mathrm{rpm}]$ | 6000 |
| Maximum encoder speed $[\mathrm{rpm}]$ | 3000 |
| Stall torque $[\mathrm{Nm}]$ | 0,80 |
| Peak torque $[\mathrm{Nm}]$ | 2,60 |
| Supply voltage $[\mathrm{V}]$ | 48 |
| Stall current $[\mathrm{Arms}]$ | 7,87 |
| Peak current $[\mathrm{Arms}]$ | 25,58 |
| Encoder resolution $[\mathrm{pulses}]$ | 2048 |

Table 5: Motors technical data

| Associated Motor | Motor 1 | Motor 2 | Motor 3 |
| :---: | :---: | :---: | :---: |
| Producer | Datorker | Datorker | Datorker |
| Model | WUT-PO 17 | WUT-PO 17 | WUT-PO 17 |
| Reduction Ratio | 50 | 80 | 50 |

Table 6: Gearboxes data


Figure 19: Robotic leg developed by CDEI - UPC

| Producer | Ingenia |
| :---: | :---: |
| Part Number | EVE-XCR-E |
| Continuous Current $[A]$ | 45 |
| Peak Current $[A]$ | 60 |
| Supply Voltage $[V]$ | $[8,80]$ |
| Communications | EtherCAT |
| Dimensions $[\mathrm{mm}]$ | $42 \times 29 \times 23,2$ |
| Weight $[\mathrm{g}]$ | 34 |

Table 7: Motor drivers technical data


Figure 20: Ingenia EVE-XCR-E driver employed to control one motor


Figure 21: Brushless motor employed in the leg to actuate a single joint

## 6 Software

The control algorithm for the robotic system were implemented using the Robot Operating System (ROS) and MATLAB\&Simulink. Simulink was able to communicate with ROS through the use of the ROS Toolbox, which provided functions for sending and receiving ROS messages and calling ROS services. Before deploying the software on the physical robotic system, simulations were performed using a physics simulator such as Gazebo, which allowed for visualization of the 3D model of the leg as it was moved by the control algorithm. These simulations included basic movements such as single joint control, as well as more complex trajectories. The highlevel control algorithm, which implemented the algorithm described in Section 4.1, has been developed in Simulink, while the low-level communication with the hardware or simulator was handled by ROS. To use the simulation environment in Gazebo, it was necessary to create a 3D model of the robotic leg in a format that Gazebo could understand, such as URDF. The process of generating the URDF file, coding the ROS nodes, and establishing the interconnection between Simulink and ROS will be described in this chapter.

### 6.1 Robotic Operating System - ROS

Robot Operating System (ROS) is a powerful and flexible software platform for building robot applications. It is composed of a set of libraries and tools that provide a wide range of functionality for robotics, including hardware abstractions, low-level device control, and high-level robot behaviors. ROS is designed to be modular and flexible, allowing developers to easily create and integrate new robot applications by building on top of the existing ROS infrastructure. It offers a range of drivers and algorithms that support a variety of hardware platforms and sensor types, making it easy to connect and control hardware devices. In addition to its core libraries and tools, ROS also provides a number of developer tools that make it easier to build and debug robot applications. These include tools for visualizing data, debugging code, and profiling performance. One of the key benefits of ROS is that it is open source, meaning that it is freely available and can be modified and extended by anyone. This makes it an ideal platform for collaborative robotics projects, as developers can freely share and build upon each other's work.


Figure 22: ROS logo

In the Robot Operating System (ROS) environment, the basic units of computation are called nodes. Each node is responsible for performing a specific, small task. In a ROS project, multiple nodes work together to perform larger, distributed computing tasks. As an example, consider a navigation robot that needs to perform motor control, perceive its environment, filter sensor data, and execute high-level intelligent control. Each of these tasks can be assigned to a separate node, and the project is assembled by connecting these nodes and allowing them to communicate with each other. If there is a need to modify part of the project in the future, it is only necessary to modify a single node, rather than the entire codebase. This modular approach to computing is particularly useful in cases where the robotic system may need to be integrated
into a larger, more complex system, such as a quadruped robot.
One key feature of ROS is that nodes can be written in different programming languages, such as $\mathrm{C}++$ or Python, and still communicate with each other. $\mathrm{C}++$ is a compiled language, which means it executes faster than Python, which is an interpreted language. For this reason, it is often preferred for real-time applications like the one described in this project, where high real time performance was required.

### 6.1.1 ROS vs ROS2

One limitation of ROS is that it is only compatible with specific versions of Ubuntu Linux. This means that it can only be installed and run on these specific versions of the operating system. To address this issue, the developers of ROS have created a new version of the framework called ROS2. ROS2 is designed to have a wider range of compatibility with different operating systems, including support for additional versions of Linux and even some non-Linux operating systems. This increased compatibility can make it easier to use ROS2 on a wider variety of hardware platforms and can also allow for more flexibility in terms of deployment options. Despite the potential benefits of using ROS2, it is still a relatively new version of the framework and may have less documentation and support available in the open-source community compared to the original version of ROS. For this reason, the version of ROS used in this project (ROS noetic) was chosen to run on Ubuntu 20.04, a version of Linux that is supported by both ROS and ROS2.

### 6.1.2 Communication between nodes and rqt_graph

Nodes can communicate each others by exchanging data in 3 ways:

1. Topics: they are useful for continuous data stream between 2 or more nodes. This paradigm is based on a publisher/subscriber mechanism in which the first node sends a message and the second receives it.
2. Services: they are useful for sending instant request(once, not in continuous way) from a node to another, by using a client/server mechanism.
3. Actions: they are useful for long term task and are also based on a client/server mechanism. They are executed asynchronously so other tasks can be performed in parallel. Actions are the core of controllers implementation in ROS. They are basically constituted by a collection of topics.

The rqt_graph tool allows users to see the relationships between nodes in a visual way, making it easier to understand how the system is structured and how data is flowing between nodes. This can be especially useful in larger, more complex systems where there may be many nodes interacting with each other in different ways. Using the rqt_graph, users can see which nodes are publishing and subscribing to which topics, as well as which nodes are providing and using services. This information can help users identify potential bottlenecks or issues in the system and can also provide insight into how the system is functioning overall.

In addition to its utility in understanding the structure and behavior of a ROS system, the rqt_graph tool can also be useful for debugging and troubleshooting issues. By visualizing the connections between nodes, users can more easily identify where problems may be occurring and can take steps to resolve them.

Overall, the rqt_graph is a valuable tool for visualizing the connections between nodes in a ROS system and can be helpful in understanding how the system is structured, identifying potential issues, and debugging problems. The rqt_graph of this project is visualized in Figure 23 and it will be explained better in the following sections.


Figure 23: rqt_graph of this project

### 6.2 Gazebo Simulator

Gazebo is a simulation tool for robotic systems that has a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. It is a standalone application that is also integrated as a ROS package. Gazebo can communicate with the ROS ecosystems through several topics, services and actions. It is very convenient to perform some simulations in a simulator like Gazebo before trying to control the real system. However it's not always true that if an algorithm works well in simulation, it will work well also in the real world. This is due to many factors, but mainly due to the complexity of real scenarios. To visualize and use a 3D model of a robot in Gazebo, it is requested to describe it in a file with the extension . urdf. Unified Robotics Description Format, URDF, is an XML specification used in academia and industry to model multibody systems such as robotic manipulator arms, mobile robotic platforms and in this case the robotic leg. It uses a tree data structure to represent the hierarchy between the links and their connections through joints. For this reason it is not the best solution to represent parallel structures.

### 6.3 Launch file

When doing a ROS project, it can happen that several nodes have to be run. Since to run each single node it's necessary to open a new terminal, it is not so comfortable to have a lot of terminal windows opened. To overcome this issue, it is possible to run the whole project by executing a .launch file. The launch utility is very useful for large projects since it permits to run and configure several nodes at the same time, do a remap of the topics, load 3D URDF models, load controllers nodes and so on. A .launch file is written in .xml format and when called it checks automatically whether the master is running or not, and if not it automatically calls it. In this project, two launch files have been used. The first, called gazebo_parallel.launch, is necessary to load the Gazebo empty world and spawn the leg 3D model. The second launch file is necessary to load all the controllers (trajectory and state controllers) as long as the ROS node necessary for controlling the simulated system. The launch files are reported in the Appendix.

### 6.4 Generating the URDF model from the 3D model

To use the physics simulator Gazebo, it was necessary to generate a 3D model of the leg in a format that Gazebo can understand. One option for doing this was to create a Unified Robot Description Format (URDF) file. A URDF file is an XML file that describes the kinematic and dynamic properties of a robot, including its joints, links, and sensors. It is used to represent the robot in simulation environments like Gazebo. To create a URDF model of the robotic leg, it was needed to specify the geometry, mass, and inertial properties of each link, as well as the joints that connect the links together. It was also requested to define any sensors or other hardware components that are mounted on the leg. Once all these elements had been defined, it was possible to use the URDF file to load the model into Gazebo and simulate the leg's behavior. Generating a URDF model of the robotic leg can be a complex process, but it is an important step in using Gazebo for simulation and testing. With a properly defined URDF model, it is possible to accurately simulate the leg's behavior and fine-tune the control algorithms before deploying them on a physical robot. The 3D model of the mechanical structure of the leg had already been developed by the CDEI and was available in .asm format. This model can be visualized and modified using a mechanical design software such as SolidWorks, as shown in Figure 24.


Figure 24: 3D model of the leg visualized in Solidworks
The Solidworks to URDF exporter (SW2URDF) is a valuable add-on for creating a URDF model of a robotic 3D structure from a Solidworks model. It can greatly simplify the process of defining the tree structure of links and joints, as well as the properties of each joint. This can save a significant amount of time compared to manually creating a URDF file, which can be a tedious and error-prone process. The SW2URDF tool allows users to define the origin and orientation of each joint, as well as the type of joint (e.g., revolute, prismatic, etc.) and the axis of rotation. It also allows users to export the resulting URDF file and STL files for the meshes of the robotic structure. This can be especially helpful for creating 3D models of complex systems with many links and joints, as it can be difficult to manually define all of these properties in a URDF file. Another advantage of using the SW2URDF tool is that it is a visual tool, which means that users
do not need to manually enter values such as the origin of a joint with respect to its parent link. Instead, they can simply use the graphical interface to define these properties, which can be much faster and more intuitive. This can help to reduce the risk of errors and ensure that the resulting URDF model is accurate and complete.
As mentioned previously, the mechanical system for the leg has a parallel structure, which may not be well represented in a tree structure like that used in a URDF file. To address this issue, initial simulations were performed using a simplified version of the 3D model that only included the three links Link1, Link2, Link3A connected by three revolute joints. This simplified model can be visualized in Gazebo in Figure 25, and its tree representation is shown in Figure 26.

A closed chain structure, such as the one used in this project for the leg, can be more accurately represented using a .sdf file rather than a .urdf file. This is because the . sdf format allows for the use of a graph data structure, which can better capture the complex kinematic relationships present in a closed chain system. However, in some cases it may be necessary or desirable to use a . urdf file instead. One approach to represent a closed chain system in a . urdf file is to use a workaround available in Gazebo. While this approach is not ideal for accurately capturing the full kinematic behavior of the system, it can be a useful option in some cases where a .sdf file is not practical or desired. The approach is the following:

1. Describe the robotic leg like in the simplified structure but adding an extra branch starting from Link2 to Link5 to represent the transmission bar. The tree representation is showed in Figure 28.
2. Add an extra fake joint of type fixed connecting $\operatorname{Link} 3 B$ to $\operatorname{Link} 5$ in order to close the tree into a graph structure. This joint description has to be written in . sdf format inside a <gazebo> tag, as following. In this case it was necessary to specify the pose of the joint which was put at the origin of Link3B and the parent and child links which the joint connects, namely Link3B and Link5:
```
<gazebo>
    <joint name="fake_joint" type="fixed">
        <pose>0 0 0 0 1.5708 0</pose>
        <parent>link_3B</parent>
        <child>link_5</child>
    </joint>
</gazebo>
```

An extra cuboidal shape has been added to the base link in order to work as a support structure. The resulting 3D Gazebo representation is shown in Figure 27. In Figure 28 it is possible to visualize the tree representation of the complete URDF model but without the fake joint described previously.


Figure 25: Simplified 3D model without the parallel structure represented in Gazebo


Figure 26: Tree representation of links connections in the simplified robotic leg description


Figure 27: 3D model of the leg including the transmission bar visualized in Gazebo


Figure 28: Tree representation of the URDF description considering the parallel structure but not fully connected

ETSEIB

### 6.5 Defining the controllers

Before implementing more complex control operations, small tasks were first performed, such as single joint movements for each joint in the model. To achieve this, it is necessary to use one or more controllers that will be used by a ROS node to communicate with Gazebo. Each controller must be defined in a file with the .yaml extension, in which the namespace and type of controller can be specified, along with other relevant parameters such as the joint names (which must match those in the URDF file) and the PID parameters for each joint. It is important to note that while the URDF model has six revolute joints, only three of them are active joints and need to be included in the controller file.
To be able to control each joint independently, it is necessary to make some modifications to the URDF file. Specifically, some Gazebo tags must be added.

- The <transmission> element has to be defined to link actuators to joints and it has be defined only for the active joints. The important information in these transmission tags are:

```
<joint name='name'>: the name must match a joint name of the URDF file.
```

<type>: the type of transmission. transmission_interface/SimpleTransmission is the only implemented interface.
<hardwareInterface>: within both the <joint> and <actuator> tags, this was able to tell the gazebo_control plugin what hardware interface to load (position, velocity or effort interfaces). Currently only position interfaces are implemented. The implementation for a single joint is the following:

```
<transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint_0">
            <hardwareInterface>PositionJointInterface</hardwareInterface>
        </joint>
        <actuator name="motor1">
            <hardwareInterface>PositionJointInterface</hardwareInterface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
</transmission>
```

- A Gazebo plugin needs to be added to the URDF to actually parse the transmission tags and load the appropriate hardware interfaces and controller manager, like this:

```
<gazebo>
    <plugin filename="libgazebo_ros_control.so" name="gazebo_control">
            <robotNamespace>/leg_controller</robotNamespace>
    </plugin>
</gazebo>
```

where the parameter <robotNamespace> has to be the same of the namespace of the controllers defined in the .yaml file

With these considerations in mind, it has been possible to define the file containing the controller description. As reported in the following:

Listing 1: ". yaml file in which the controllers are defined"

```
leg_controller:
    joint_traj_controller_parallel:
        type: position_controllers/JointTrajectoryController
        joints:
            - joint_0
            - joint_1
            - joint_2
        gains:
            joint_0: {p: 100.0, i: 90.0, d: 3.0}
            joint_1: {p: 100.0, i: 90.0, d: 3.0}
            joint_2: {p: 60.0, i: 200.0, d: 3.0}
        constraints:
            goal_time: 0.6
            stopped_velocity_tolerance: 0.05
            joint_0: {trajectory: 0.1, goal: 0.1}
            joint_1: {trajectory: 0.1, goal: 0.1}
            joint_2: {trajectory: 0.1, goal: 0.1}
    joint_state_controller:
            type: joint_state_controller/JointStateController
            joints:
            - joint_0
            - joint_1
            - joint_2
            publish_rate: 20
```

All the controllers defined in this file belong to the /leg_controller namespace. In the second line it is defined the controller name/joint_traj_controller_parallel which belongs to the family of type position_controllers/JointTrajectoryController. It is possible to notice also the joints involved in this control and their relative PID gains. As mentioned before, only the three active joints of our system have to be included in this description.
The second controller joint_state_controller has been defined in order to allow Gazebo to publish a topic containing the current joint states in the ROS network. This will be very useful later to have a feedback in the control process. The joint states will be contained in a topic named /leg_controller/joint_states and will be used by Simulink as feedback, as discussed in the next sections.

### 6.6 MATLAB\&Simulink

MATLAB and Simulink are popular engineering and research tools. MATLAB is a programming language for numerical computation and data analysis, with built-in functions and toolboxes. Simulink is a model-based graphical simulation and design tool for building and simulating complex systems, like control systems, using block diagrams. In this project, we have used MATLAB and Simulink to develop the control algorithms for the robotic leg. By using these tools, we were able to design and test our algorithms in a simulated environment before implementing them on the real hardware. This allowed us to iterate and optimize our designs, and to ensure that they would behave as expected when deployed on the robotic leg. In particular, we used Simulink to build the high-level control logic, which was responsible for generating the desired joint positions and velocities based on the desired trajectory of the leg.

To facilitate the integration of our control algorithms with the hardware, we used the ROS Toolbox, which is a collection of tools and libraries that allow Simulink models to communicate with ROS nodes. This enabled us to easily send and receive data between our Simulink models and the ROS nodes that were responsible for controlling the robotic leg. We also used Gazebo to visualize the 3D model of the leg as it moved according to our control algorithms. This allowed us to see the performance of our algorithms in a realistic, 3D environment, and to fine-tune our designs as needed. The hierarchy between these tools that we used is reported in Figure 30. Alternatively to Gazebo, ROS can also communicate with the hardware of the mechanical system in order to control the robotic leg in an experimental scenario.

Overall, the use of MATLAB and Simulink has been crucial to the success of this project. By using these tools, we were able to design and test our control algorithms in a flexible and efficient manner, and to quickly prototype and iterate on our designs. The integration with ROS and Gazebo has also been valuable, as it has allowed us to easily connect our control algorithms to the hardware and to visualize the performance of our algorithms in a realistic environment. In the following sections, we will describe in detail the methods and techniques that we used to develop the control algorithms for the robotic leg, and we will present the results of our simulations and experiments. The Figure 29 represents the implementation in Simulink of the Jacobian Inverse control scheme inspired by the algorithm discussed in the literature and reported in Figure 18.

This scheme is useful for performing simulations in Simulink to see if the algorithm performs as it has to and to fine-tune the controller. More complex schemes including the ROS integration will be showed later. The main parts to be noticed in this scheme are the following:

- The Saturation block has been introduced to limit the joint positions matching to the physical limit of each joint in the mechanical system.
- The Jacobian_Inverse block useful to compute the inverse of the Jacobian matrix in each simulation step: it takes as input the current joint configuration $\boldsymbol{q}=\left(\boldsymbol{q}_{1}, \boldsymbol{q}_{2}, \boldsymbol{q}_{3}\right)$ and it has been implemented as a MATLAB function.
- The Forward Kinematics block which function is to compute the cartesian position $x_{e}$ (expressed in meters [m]) of the end effector of the robot given the current joint configuration $q=\left(\boldsymbol{q}_{1}, \boldsymbol{q}_{2}, \boldsymbol{q}_{3}\right)$. Also this block has been implemented as a MATLAB function.
- The input trajectory has to be defined in the three coordinates of the cartesian space and
then put inside a Multiplexer block in order to be interpreted as a 3D column vector by Simulink.

In this case it is represented a very simple trajectory where the foot of the leg oscillates along a line in the $z$ direction. More complex trajectories and their generation process will be explained and discussed in the following sections.


Figure 29: Simple Simulink implementation of the Jacobian Inverse algorithm

### 6.6.1 Integration between Simulink and ROS

The integration between Simulink and the Robot Operating System (ROS) is facilitated by the ROS Toolbox, which is a collection of tools and libraries that enable Simulink models to communicate with ROS nodes. Using the ROS Toolbox, users can send and receive data between Simulink models and ROS nodes, and can use ROS services and parameters within Simulink models. This allows users to easily incorporate sensor and actuator data from the robot into their Simulink models, and to control the robot's behavior based on the outputs of their Simulink models. The ROS Toolbox also includes a set of pre-built Simulink blocks for common ROS functionality, such as publishing and subscribing to ROS topics and calling ROS services. The modified Simulink schematic is shown in Figure 31 where the integration with ROS environment has been added. It can be noticed that now the vector of joints positions $\boldsymbol{q}=\left(\boldsymbol{q}_{\boldsymbol{1}}, \boldsymbol{q}_{\boldsymbol{2}}, \boldsymbol{q}_{\boldsymbol{3}}\right)^{T}$ has been published into a rostopic. As shown better in Figure 32 the vector $\boldsymbol{q}=\left(\boldsymbol{q}_{\mathbf{1}}, \boldsymbol{q}_{\mathbf{2}}, \boldsymbol{q}_{\mathbf{3}}\right)^{\boldsymbol{T}}$ has been put into an object of type geometry_msgs/Pose with the correspondence $\boldsymbol{q}=\left(\boldsymbol{q}_{\boldsymbol{1}}, \boldsymbol{q}_{\boldsymbol{2}}, \boldsymbol{q}_{\boldsymbol{3}}\right)^{\boldsymbol{T}}$ $=$ (Position.X , Position.Y , Position.Z) and then published inside a topic whose name is /joint_state_publisher. The feedback in this case comes from an external source which can be Gazebo joint states (if we are performing simulations with the digital twin) or the incremental encoders of the motors(if we want to use the experimental system). As shown in Figures 31,33 whenever a new rostopic of the type /leg_controller/joint_states is published, the current joint states vector $\boldsymbol{q}=\left(\boldsymbol{q}_{1}, \boldsymbol{q}_{2}, \boldsymbol{q}_{3}\right)^{\boldsymbol{T}}$ can be read and used as feedback by the control loop. It is important in this case to set some additional parameters in Simulink Settings, among the others:

- Solver Type set to Fixed-step
- Fixed-step size set to 0.001 s . This time step has to be coherent with the ROS rate of all the involved nodes in the network. The publisher and subscriber objects must have the same rate.

A good way to visualize the interconnection between Simulink, ROS and Gazebo is to look a the rqt_graph reported in Figure 34. As explained before, the node /joint_control_node receives the topic /joint_state_publisher from the Simulink model, containing the requested joint positions. It then publishes these values by using the action controller described in Section 6.5 /leg_controller/joint_traj_controller_parallel/follow_joint_trajectory which will be used by the URDF model to move. Gazebo consequently can publish the current joint states inside the topic /leg_controller/joint_states which will be used as feedback by the Simulink model. In this sense we have a closed loop control system. For the experimental setup, we utilized some ROS nodes written by the company Beta Robots to handle the low-level communication with the hardware. The software is designed to provide a communication interface between the control algorithm and the robotic leg's hardware, allowing us to send commands and receive feedback from the robot. This software enabled us to easily interface with the robot's motors and to control the robot's movements with high precision. The detailed description of this software is not part of this works as at the high-level side, the ROS nodes topology is the same as the one used during the simulations in Gazebo. By using the same node topology for both the simulations and the experiments, we were able to easily transfer the control algorithm from the simulation environment to the experimental setup. This greatly simplified the process of testing the algorithm on the real robot and allowed us to quickly evaluate its performance in a real-world scenario.


Figure 30: Hierarchy representing the communication between MATLAB\&Simulink, ROS, Gazebo simulator and the real hardware


Figure 31: Implementation of the Jacobian Inverse algorithm interfacing Simulink with ROS through routines of ROS-Toolbox


Figure 32: Block scheme for publishing a topic in ROS network from Simulink


Figure 33: Block scheme for subscribing to a topic in ROS network from Simulink


Figure 34: rqt_graph of the whole control system

### 6.7 Operational space trajectory generation

In this project, a bioinspired approach was adopted for the real operational space trajectory, with the specific aim of replicating the movement of a dog's leg. There are various methods
that can be employed for this purpose, however, utilizing pre-existing motion capture data is considered the most simple and efficient method. Motion capture, or mocap, is a technique that employs sensors and cameras to track and record the movement of an object or person in 3D space. This data can subsequently be used to produce detailed and precise animations or simulations of the movement. In this project, motion capture data of a dog's leg movement was employed as a benchmark to replicate the movement in the robotic system. There are several motion capture (mocap) datasets available on the internet that can be accessed for free. In this case, we chose to use one of these datasets to obtain the trajectory data for the project. However, there is an issue with the dataset as it represents the movement of a dog's foot with respect to the reference frame of the camera, rather than in the dog's body reference frame. As a result, the final trajectory of the dog's movement, as shown in Figure 35 may not be accurate or correctly representative of the movement. To use this data in our project it was necessary first to represent the trajectory points with respect to the leg's shoulder reference frame. To this purpose, when dealing with motion capture data, at least three markers are put in the body of the dog or on the shoulders: this way it is possible to perform a change of coordinate system in order to represent the trajectory points with respect to the dog's body. In the following section it is explained how this procedure was performed and which are the obtained results.


Figure 35: Dog's foot trajectory with respect to camera reference frame

ETSEIB


Figure 36: Example of markers distribution on the body of a dog for motion capture highlighting the relevant markers used in this project [Source: DeepAI]

Let $\boldsymbol{P}^{C}$ be the vector representing the three cartesian coordinates of the foot marker with respect to the Camera Reference Frame $\left(R F_{C}\right)$.
Let $\boldsymbol{L} \boldsymbol{B}^{C}, \boldsymbol{L} \boldsymbol{F}^{C}, \boldsymbol{F} \boldsymbol{R}^{C}$ be respectively the Left-Back, Left-Forward and the Right-Forward markers coordinates represented with respect to $R F_{C}$, as shown in Figure 36.
We want to perform a roto-translation of the data points, in order to have the origin of the Robot Reference Frame $\left(R F_{R}\right)$ in $\boldsymbol{L B}$. To this purpose it is necessary to build a reference frame having its origin in $\boldsymbol{L B}$. Let:

$$
\begin{equation*}
T=L B^{C} \tag{34}
\end{equation*}
$$

be the translation vector from the $R F_{C}$ to the $R F_{R}$. Let:

$$
\begin{equation*}
x_{R}^{C}=L F^{C}-L B^{C} \tag{35}
\end{equation*}
$$

be the first vector of the $R F_{R}$. Let:

$$
\begin{equation*}
a=L F^{C}-F R^{C} \tag{36}
\end{equation*}
$$

be a coplanar vector to $x_{R}^{C}$ in the robot body. Then to obtain the $z_{R}^{C}$ normal to both the former vectors, the following holds:

$$
\begin{equation*}
z_{R}^{C}=x_{R}^{C} \otimes a \tag{37}
\end{equation*}
$$

To complete the right handed triplet it is sufficient to add the $y_{R}^{C}$ vector, given by the outer product:

ETSEIB

$$
\begin{equation*}
y_{R}^{C}=z_{R}^{C} \otimes x_{R}^{C} \tag{38}
\end{equation*}
$$

At this point we have obtained a orthogonal basis, but to achieve an orthonormal basis a normalization process is needed, so:

$$
\begin{gather*}
x_{R}^{C}=\frac{x_{R}^{C}}{\left\|x_{R}^{C}\right\|}  \tag{39}\\
y_{R}^{C}=\frac{y_{R}^{C}}{\left\|y_{R}^{C}\right\|}  \tag{40}\\
z_{R}^{C}=\frac{z_{R}^{C}}{\left\|z_{R}^{C}\right\|} \tag{41}
\end{gather*}
$$

The following then represents the rotation matrix from $R F_{R}$ to $R F_{C}$ :

$$
R_{R}^{C}=\left[\begin{array}{lll}
x_{\boldsymbol{R}}^{C} & y_{\boldsymbol{R}}^{C} & z_{\boldsymbol{R}}^{C} \tag{42}
\end{array}\right]
$$

and consequently the homogeneous transformation matrix from $R F_{R}$ to $R F_{C}$ is:

$$
M_{R}^{C}=\left[\begin{array}{cc}
\boldsymbol{R}_{R}^{C} & \boldsymbol{T}  \tag{43}\\
\mathbf{0}^{T} & 1
\end{array}\right] \in S E(3)
$$

To represent then the point $P$ with respect to the $R F_{R}$ then the following holds:

$$
\begin{equation*}
P^{R}=M_{C}^{R} P^{C}=\left(M_{R}^{C}\right)^{-1} P^{C} \tag{44}
\end{equation*}
$$

The previous process has been iterated for every trajectory point in order to obtain the foot trajectory represented with respect to the robot body, as shown in Figure 38. The trajectory is represented only into the $(x, z)$-plane and, as expected, it is a closed curve. After this preprocessing in MATLAB, the trajectory points can be stored into a Structure with time to be consequently used by a Simulink block as a Source Signal $x_{d}$ as shown in Figure 37. The $y$ coordinate is kept constant to the value $l_{1, A}$ for design choice: this way we are sure that the leg can simulate well a dog walking movement.


Figure 37: Generation of the source trajectory in Simulink to simulate the movement of a real dog's toe


Figure 38: Dog foot trajectory represented with respect to the robot body reference frame

## 7 Simulation results

Before deploying the Jacobian Inverse algorithm into an experimental system, it was important to perform simulations in order to check the correctness of the inverse kinematic algorithm and to fine-tune the controller. This ensures that the algorithm is working as expected and that any potential issues can be identified and resolved before the algorithm is implemented in a realworld scenario. To accomplish this, the control system was first tested in Simulink without any interconnection with ROS. This allowed for the algorithm to be tested in a controlled environment, where the simulation results could be easily analyzed and any issues could be quickly identified and addressed. By testing the algorithm in Simulink, it was possible to verify that the Jacobian Inverse algorithm was working correctly and that the controller was able to accurately control the system. Once the algorithm had been tested in Simulink, a second simulation was performed by simulating a real scenario in Gazebo. This allowed for the algorithm to be tested in a more realistic environment, where the system would be subjected to the same types of disturbances and noise that it would encounter in a real-world scenario. By simulating a real scenario in Gazebo, it was possible to fine-tune the controller and to ensure that the algorithm would perform well in a real-world scenario. The simulation results will be presented and discussed in this section. The results of the Simulink simulation will be compared with the results of the Gazebo simulation to ensure that the algorithm works correctly in both environments. Any discrepancies between the simulation results will be analyzed to identify the cause and to determine the necessary adjustments to the algorithm or controller.

The evolution of the norm of the error between $x_{e}$ and $x_{d}$ is shown in Figure 40. As discussed in Section 4.1 the error function has a converging exponential behaviour to zero with convergence rate that depends on the controller matrix $\boldsymbol{K}$. In this case the tuning of the matrix $\boldsymbol{K}$ was done in an empirical way, taking into account that it had to be positive definite. The chosen matrix guarantees that the error converges rapidly to 0 and at the same time that the joint velocities assume limited values, as reported in Figure 39. It has to be noticed that the joint velocities are limited also due to the fact that the system never falls in a singular configuration or near a singularity, due to the design choices discussed in section 4.2. Consequently the joint positions assume limited values. The $y$-coordinate's initial condition of the leg's foot had been set to $l_{1, A}$, which is the same constant setpoint chosen for the $y$-coordinate of the input trajectory. Since this coordinate is controlled only by Joint 1, this joint never moves. We cannot say the same thing for the Gazebo simulation since when the leg's 3D model is spawned, due to gravity effect the leg's foot won't have $y$-coordinate's initial condition equal to $l_{1, A}$. For this reason there is a small amount of time in which the Joint 1 is being actuated to achieve the desired $y$-coordinate. At this point the $q_{1}$ will reach a steady state value different from zero, since the 0 point is calculated as the initial joint position when Gazebo is started. Also in this case the joint velocities and positions are kept limited as shown in Figure 42. The operational space error, as displayed in Figure 43 , converges to zero as expected, but not as good as in the Simulink simulation. Indeed the maximum absolute displacement at steady state is $0.024 m$.

The comparison between the desired and achieved trajectory in Gazebo simulation is shown in Figure 44 which, as expected is worse than the simulated result in Simulink, shown in Figure 41.



Figure 39: Joint positions and velocities behaviour in Simulink simulation


Figure 40: Error norm in cartesian space in Simulink simulation


Figure 41: Comparison between desired trajectory $x_{d}$ and followed trajectory $x_{e}$ visualized in the $(x, z)$-plane

ETSEIB


Joint velocities, Gazebo simulation


Figure 42: Joint positions and velocities behaviour in Gazebo simulation


Figure 43: Error norm in cartesian space in Gazebo simulation


Figure 44: Comparison between desired trajectory $x_{d}$ and followed trajectory $x_{e}$ visualized in the $(x, z)$-plane, during Gazebo simulation

ETSEIB

## 8 Experimental results

After performing the simulations in Simulink and in a more realistic setup through Gazebo, we were ready to test the kinematic control algorithm in the experimental system present at the CDEI. This experimental system allowed us to test the algorithm in a real-world scenario and evaluate its performance under various conditions. The results of these experiments were recorded and analyzed in detail to gain insights into the algorithm's behavior and performance. In the following section, the experimental results will be presented and discussed in detail. These results will be compared with the simulation results obtained in Simulink and Gazebo to evaluate the accuracy and validity of the simulation models. The comparison of simulation and experimental results will also allow us to identify any discrepancies or limitations of the algorithm and suggest modifications or improvements for future work.

In Figure 46, the error's norm evolution between the current end-effector position $x_{e}$ and the desired end-effector position $x_{d}$ is shown. As expected, it shows a converging behavior by decreasing exponentially to zero. It can be noticed that even at steady state, the error has some small oscillations due to external disturbances to which a real system is always subjected. However, after an initial transient time, the leg achieved to follow the desired trajectory (as shown in Figure 47) with a maximum absolute error of 4 mm . It is surprising to notice that the experimental system performs way better than the Gazebo simulation.

It has to be noticed from Figure 45 that the joint positions and velocities are kept limited inside the motor limits imposed by the algorithm and the experimental system. The evolution of $q_{1}$ deserves particular attention as it is always constant to 0 . This is because its electrical driver didn't work and for design choices, it was decided to not use its related motor. With a blocking mechanism, the leg was put in a configuration such that its $y$-coordinate had been kept constant to $l_{1, A}$ and the control was performed only by commanding Joint2 and Joint3. For this reason, it makes more sense to visualize the followed trajectory in the $(x, z)$-plane as shown in Figure 47.

The results of the experiments demonstrate that the proposed algorithm is able to achieve a satisfactory performance in the experimental system, despite the limitations imposed by the malfunctioning motor and the presence of external disturbances. The comparison of simulation results obtained in Simulink and Gazebo with the experimental results also allowed us to evaluate the accuracy and validity of the simulation models, and to identify any discrepancies or limitations of the algorithm. These findings have important implications for the design and optimization of similar systems in the future, and lay the foundation for further research in this area. Furthermore, the results also demonstrate that the proposed kinematic control algorithm is robust and able to perform well in real-world scenarios, despite the presence of external disturbances and limitations imposed by the experimental system.


Joint velocities, experimental results


Figure 45: Joint positions and velocities during experimental test


Figure 46: Error norm in cartesian space during experimental test


Figure 47: Comparison between desired trajectory $x_{d}$ and achieved trajectory $x_{e}$ visualized in the $(x, z)$-plane, during experimental simulation

ETSEIB

## 9 Temporary planning and costs

In this section an analysis of the cost of the project will be presented. The analysis will be divided in three parts: cost of workers, energy cost, cost of the software used.

### 9.1 Cost of workers

The tasks described in this report were carried out by the supervisor Alba Perez Gracia and the student Roberto Carta. According to the criteria followed in other projects carried out at the CDEI, it is estimated that the hourly cost of the project manager is $58 € / h$ and that of a junior project engineer is $38 € / h$. The time used by the student is estimated considering that one ECTS requires 25 hours of dedication and this work is 30 ECTS. The hours dedicated by the tutor have been counted directly. These points are summarized in the following table.

| Position | Hourly Cost $[€ / h]$ | Dedicated hours | Price $[€]$ |
| :---: | :---: | :---: | :---: |
| Supervisor | 58 | 30 | 1.740 |
| Student | 38 | 750 | 28.500 |
| Total |  |  | $\mathbf{3 0 . 2 4 0}$ |

Table 8: Cost of personal

### 9.2 Energy Cost

Energy cost is understood as any expense associated with the energy consumption of the equipment used throughout this work.The energy consumed that is taken into account is primarily that consumed by the computer equipment used and the lighting used in the work spaces. The total associated cost is calculated using the following formula:

$$
\begin{equation*}
\operatorname{Cost}[€]=\text { Consumption }[k W] \cdot \text { FunctioningHours }[h] \cdot \text { ElectricityPrice }[€ / k W h] \tag{45}
\end{equation*}
$$

According to the data [16] the average electricity price in Spain in 2022 has been $0,346 € / k W h$.
To this energy consumption, also the consumption of the Power Supply for the experimental setup has to be taken into account. Supposing that the real system has been tested during the last 2 weeks of the project it has been functioning for an average of $80 h$, with a medium consumption of $8 A$ per motor, considering that we only moved two motors. Moreover the supply voltage is 48 V . In the following table the total energy cost is reported.

| Element | Consumption $[k W]$ | Functioning hours $[h]$ | Price $[€]$ |
| :---: | :---: | :---: | :---: |
| Desktop PC | 0,350 | 750 | 90,83 |
| PC Screen $(\times 2)$ | 0,100 | 750 | 25,95 |
| Laboratory Illumination | $10 \times 0,036$ | 750 | 93,42 |
| Leg power supply | 0,768 | 80 | 212,45 |
| Total |  |  | $\mathbf{4 2 2 , 6 5}$ |

Table 9: Cost of energy

| Software | Price $[€]$ |
| :---: | :---: |
| Ubuntu | Free |
| Windows 10 | 100 |
| MATLAB\&Simulink | 500 |
| Solidworks | 99 |
| Overleaf | Free |
| ROS software by Beta | 2.000 |
| Total | $\mathbf{2 . 6 9 9}$ |

Table 10: Cost of software licenses of the project

### 9.3 Software licenses cost

### 9.4 Total cost of the project

To get the total cost of the project all the partial prices obtained in the previous paragraphs have to be summed.

| Software | Price $[€]$ |
| :---: | :---: |
| Sum of the cost | $33.361,65$ |
| VAT $(21 \%)$ | $7.005,95$ |
| Total cost | $\mathbf{4 0 . 3 6 7 , 6 0}$ |

Table 11: Sum of the cost of the project

Eventually, the total cost of this Master Thesis Project is $40.367,60 €$.

## 10 Environmental Impact

In 2022 the average emissions of Spain were $195 \mathrm{~g} \mathrm{CO}_{2} e q / k W h$. [17]
Based on this data and recalling the analysis made in Section 9.2, the following table reports the $\mathrm{CO}_{2}$ emission for this project.

| Element | Consumption $[\mathrm{kW}]$ | Functioning hours $[\mathrm{h}]$ | $\mathrm{CO}_{2}$ footprint $[\mathrm{kg}]$ |
| :---: | :---: | :---: | :---: |
| Desktop PC | 0,350 | 750 | 51,190 |
| PC Screen (x2) | 0,100 | 750 | 14,625 |
| Laboratory Illumination | $10 \times 0,036$ | 750 | 52,650 |
| Leg power supply | 0,768 | 80 | 119,733 |
| Total |  |  | $\mathbf{2 3 8 , 1 9 8}$ |

Table 12: $\mathrm{CO}_{2}$ emission for the whole project
In conclusion the total $\mathrm{CO}_{2}$ emission for this project has been $238,198 \mathrm{~kg}$.

## 11 Conclusions

In this work, it was established that it was possible to guide the robotic leg to execute specific trajectories like a bioinspired trajectory similar to the one of a dog's leg. The outcomes obtained from this work represent a significant contribution to the field of robotic leg control and can serve as a foundation for the implementation of more advanced control algorithms in the future. However, it should also be acknowledged that the performance of the robotic system was not optimal due to the utilization of only kinematic control. Kinematic control, while having its own advantages, has several limitations, including the inability to track rapidly changing trajectories. In order to improve the performance of the robotic leg, one of the future steps for this project would be to acquire the dynamic model of the system and to implement a dynamic control capable of interacting also with the environment.

The student learned how to create a detailed representation of the robotic leg in URDF, including the links, joints, and sensors. Additionally, he gained experience in working with the ROS environment, which is widely used in robotics research and industry, by using ROS tools and libraries to control the robotic leg, such as the ROS controllers, rqt utility and URDF parser. The skills and knowledge acquired in this process are valuable assets that the student will be able to apply in future projects and career opportunities in the field of robotics.

Additionally, using Simulink to simulate the system behavior prior to testing the control algorithm in the experimental setup was useful in fine-tuning the control law. This allowed for a more thorough testing and optimization of the control algorithm before implementing it on the physical robotic leg. However, it should be noted that the kinematic control algorithm should be embedded in a C++ ROS node to improve the performance of the real-time software. This is mainly due to the latency present in the ROS network when running MATLAB\&Simulink. An alternative could be to run Simulink on a different computer connected through SSH with the main ROS network.

The use of already existing data to obtain optimized bio-inspired trajectories was also useful in testing the control scheme. This allowed for a more efficient and accurate testing of the control algorithm and provided valuable insights into the performance of the robotic leg. A future step is to make the system more robust to changes in terrain configuration by using a reinforcement learning approach to automatically tune the control system and follow non-predefined trajectories. This would allow the robotic leg to adapt to different environments and improve its overall performance.

The unavailability of the motor that controlled the Joint1 due to communication issues with its electronic driver had an adverse effect on the system's performance. To achieve more complex tasks and integrate the leg into a quadruped, the functioning of this motor is crucial, and thus, the electronic driver needs to be repaired. Nonetheless, in the context of the specific task that was being pursued in this work, this limitation had a positive impact on the trajectory tracking performance.

## Bibliography

[1] Bartomeu Costa Prats, Disseny mecànic d'una extremitat d'un robot quadrúpede, UPC, Escola Tècnica Superior d'Enginyeria Industrial de Barcelona, Departament d'Enginyeria Mecànica, May 2022
[2] Jordi Gené-Molaa, Ricardo Sanz-Cortiellaa, Joan R. Rosell-Poloa, Josep-Ramon Morros, Javier Ruiz-Hidalgob, Verónica Vilaplanab, Eduard Gregorio Fruit detection and 3D location using instance segmentation neural networks and structure-from-motion photogrammetry, Computers and Electronics in Agriculture Volume 169, February 2020, 105165, https://doi.org/10.1016/j.compag.2019.105165
[3] Sandilya Sai Garimella and Shai Revzen, Dandelion-Picking Legged Robot, arXiv:2112.05383v1 10 Dec 2021
[4] Míolajczyк, T.; Mikolajewska, E.; Al-Shuka, H.F.N.; Malinowski, T.; Kıodowski, A.; Pimenov, D.Y.; Paczkowski, T.; Hu, F.; Giasin, K.; Mikoॄajewski, D.; et al. Recent Advances in Bipedal Walking Robots: Review of Gait, Drive, Sensors and Control Systems, Sensors 2022, 22, 4440. https://doi.org/10.3390/s22124440
[5] Ganesh Kumar K. Pushparaj Mani Pathak, Dynamic modelling simulation of a four legged jumping robot with compliant legs, Robotics and Autonomous Systems Volume 61, Issue 3, March 2013, Pages 221-228, https://doi.org/10.1016/j.robot.2012.09.025
[6] Spot, the Boston Dynamics quadruped robot, https://robots.ieee.org/robots/ spotmini/\#:~: text=Spot\%20is\%20a\%20compact\%2C\%20nimble, also\%20fetch\%20you\% $20 a \% 20$ drink.
[7] ANYmal, Robotic Systems Lab, https://rsl.ethz.ch/robots-media/anymal.html
[8] Xiaomi Cyberdog: where to buy, features and reviews, https://xiaomipedia.com/en/p/ cyber-dog/
[9] Animator Notebook, A GUIDE TO QUADRUPEDS' GAITS, https://www. animatornotebook.com/learn/quadrupeds-gaits
[10] Wikipedia Horse gait, https://en.wikipedia.org/wiki/Horse_gait
[11] Yanran Ding, Abhishek Pandala, and Hae-Won Park Real-time Model Predictive Control for Versatile Dynamic Motions in Quadrupedal Robots 2019 International Conference on

Robotics and Automation (ICRA), Palais des congres de Montreal, Montreal, Canada, May 20-24, 2019
[12] Mohsen Sombolestan, Yiyu Chen and Quan Nguyen, Adaptive Force-based Control for Legged Robots, adaparXiv: 2011.06236v4, 15 Dec 2021
[13] Juan Carlos Arevalo and Elena Garcia Impedance Control for Legged Robots: An Insight Into the Concepts Involved IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART C: APPLICATIONS AND REVIEWS, VOL. 42, NO. 6, NOVEMBER 2012
[14] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker and Sergey Levine Learning to Walk via Deep Reinforcement Learning, arXiv:1812.11103v3, 19 Jun 2019
[15] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Giuseppe Oriolo, Robotics Modelling, Planning and Control, Springer London, 2010, ISSN 1439-2232
[16] GlobalPetrolPrices.com, Spain electricity prices, https://www.globalpetrolprices.com/ Spain/electricity_prices/
[17] NOWTRICITY.сом, CO2 emissions per kWh in Spain in 2022, https://www.nowtricity. com/country/spain/\#:~:text=In\%202022\%20the\%20average\%20emissions, Coal\% 20usage\% $\% 20$ was $\% 203.0 \% 25$.

## A Appendix

## A. 1 URDF file of the robotic leg

In the following the .urdf file containing the description of the 3D model of the leg is reported

```
<?xml version="1.0" encoding="utf-8"?>
<robot
    name="leg_new_links">
        <gazebo>
        <self_collide>false</self_collide>
        </gazebo>
<link
    name="base_link">
    <inertial>
            <origin
                xyz="0.350003733089083 0.451715069318971 0.405739623932018"
            rpy="0 0 0" />
        <mass
            value="9752.33431526145" />
        <inertia
            ixx="1198.37406604531"
            ixy="2.1247020207493E-05"
            ixz="1.68443139056609E-07"
            iyy="933.097049194367"
            iyz="-2.78321482694158E-07"
            izz="1061.59566769288" />
        </inertial>
        <visual>
            <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
                filename="package://robotic_leg/stl_files/base_link.STL" />
        </geometry>
        <material
            name="">
            <color
                    rgba="0.792156862745098 0.819607843137255 0.933333333333333 0.2" />
        </material>
        </visual>
        <collision>
            <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
                            filename="package://robotic_leg/stl_files/base_link.STL" />
        </geometry>
        </collision>
    </link>
    <link
        name="link_1">
        <inertial>
```

```
    <origin
        xyz="0.0254057149058779 2.86714343646849E-05 -0.0532471017221783"
        rpy="0 0 0" />
    <mass
        value="1.49501478481733" />
    <inertia
        ixx="0.00112894740597378"
        ixy="4.29924401856194E-07"
        ixz="2.11269028176045E-05"
        iyy="0.000724319253480805"
        iyz="2.78321942147056E-07"
        izz="0.000774164434218605" />
</inertial>
<visual>
    <origin
        xyz="0 0 0"
        rpy="0 0 0" />
    <geometry>
        <mesh
            filename="package://robotic_leg/stl_files/link_1.STL" />
    </geometry>
    <material
        name="">
        <color
            rgba="0.784313725490196 0.392156862745098 0 1" />
        </material>
    </visual>
    <collision>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_1.STL" />
        </geometry>
    </collision>
</link>
<joint
    name="joint_0"
    type="revolute">
    <origin
        xyz="0.398969739571247 -0.055 0.748315799084487"
        rpy="1.5707963267949 0 -1.5707963267949" />
    <parent
        link="base_link" />
    <child
        link="link_1" />
    <axis
        xyz="0 0 -1" />
    <limit
        lower="-3.14"
        upper="0.3"
        effort="500"
        velocity="1" />
    <dynamics
```

```
    friction="1" />
</joint>
<link
    name="link_2">
    <inertial>
        <origin
            xyz="0.021994054739837 7.48225623309548E-05 0.0480827458515721"
            rpy="0 0 0" />
        <mass
            value="1.69963884406675" />
        <inertia
            ixx="0.000833537569151538"
            ixy="-6.40943215662161E-05"
            ixz="-4.12309422837278E-06"
            iyy="0.00285881134253153"
            iyz="-9.5565990087917E-07"
            izz="0.00336502559048854" />
    </inertial>
    <visual>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_2.STL" />
        </geometry>
        <material
            name="">
            <color
            rgba="0.784313725490196 0.392156862745098 0 1" />
        </material>
    </visual>
    <collision>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
                filename="package://robotic_leg/stl_files/link_2.STL" />
        </geometry>
    </collision>
    </link>
    <joint
    name="joint_1"
    type="revolute">
    <origin
        xyz="0.05 0 -0.055"
        rpy="-1.5742 0 -1.5729" />
    <parent
        link="link_1" />
    <child
        link="link_2" />
    <axis
        xyz="0.0016667 0.002075 1" />
    <limit
```

```
    lower="-1.57"
    upper="1.57"
    effort="50"
    velocity="1" />
    <dynamics
        friction="1" />
</joint>
<link
    name="link_3A">
    <inertial>
        <origin
            xyz="0.103360504784427 1.19571440593269E-05 -0.0338432023878606"
            rpy="0 0 0" />
        <mass
            value="1.78164435082701" />
        <inertia
            ixx="0.000898120867166115"
            ixy="-8.73505939173291E-08"
            ixz="-0.000171027299674959"
            iyy="0.0137626294724966"
            iyz="-3.82341521752038E-08"
            izz="0.014208208187679" />
    </inertial>
    <visual>
            <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
                filename="package://robotic_leg/stl_files/link_3A.STL" />
        </geometry>
        <material
            name="">
            <color
                rgba="0.792156862745098 0.819607843137255 0.933333333333333 1" />
        </material>
    </visual>
    <collision>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
                filename="package://robotic_leg/stl_files/link_3A.STL" />
        </geometry>
    </collision>
</link>
<joint
    name="joint_C"
    type="revolute">
    <origin
        xyz="0.299896250144386-0.000129690447242911 0.0624997786373967"
        rpy="0.00149682425123199 0.00220071085438925 -1.65019819468379" />
    <parent
        link="link_2" />
```

```
    <child
        link="link_3A" />
    <axis
        xyz="0 0 -1" />
    <limit
        lower="-3.14"
        upper="3.14"
        effort="50"
        velocity="1" />
    <dynamics
        friction="1" />
</joint>
<link
    name="link_3B">
    <inertial>
        <origin
            xyz="0.000667581207541401 -0.00827381775791791 -0.0057577400836321"
            rpy="0 0 0" />
        <mass
            value="0.00730013217038549" />
        <inertia
            ixx="1.37042312330285E-06"
            ixy="7.5589254691122E-08"
            ixz="6.64384519026395E-09"
            iyy="4.39689862244394E-07"
            iyz="-8.23419788980468E-08"
            izz="1.43617972296875E-06" />
    </inertial>
    <visual>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_3B.STL" />
        </geometry>
        <material
            name="">
            <color
                rgba="0.0941176470588235 0.0941176470588235 0.0941176470588235 1" />
        </material>
    </visual>
    <collision>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
                filename="package://robotic_leg/stl_files/link_3B.STL" />
        </geometry>
    </collision>
</link>
<joint
    name="joint_B"
    type="revolute">
```

```
    <origin
    xyz="-0.08500000000000587 0 -0.0109999999999972"
    rpy="0 0 0" />
<parent
    link="link_3A" />
<child
    link="link_3B" />
<axis
    xyz="0 0 1" />
<limit
    lower="-3.14"
    upper="3.14"
    effort="50"
    velocity="1" />
<dynamics
    friction="1" />
</joint>
<link
name="link_4">
<inertial>
        <origin
        xyz="0.00468661861609586 0.0394322555805247 -0.00194756637030633"
        rpy="0 0 0" />
    <mass
        value="0.0406325520716012" />
    <inertia
        ixx="4.04489474912658E-05"
        ixy="-3.57255553323355E-06"
        ixz="-1.55782385850138E-08"
        iyy="1.08278091914613E-05"
        iyz="-5.95465502730334E-09"
        izz="4.97956869476247E-05" />
    </inertial>
    <visual>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_4.STL" />
        </geometry>
        <material
            name="">
            <color
            rgba="0.784313725490196 0.392156862745098 0 1" />
        </material>
    </visual>
<collision>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
    <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_4.STL" />
    </geometry>
```

    ETSEIB
    ```
</collision>
```

</link>
<joint
    name="joint_2"
    type="revolute">
<origin
$x y z="-4.83332662033556 \mathrm{E}-05-6.01763675222231 \mathrm{E}-050.0289998972877529$ "
rpy="0.00207505152704244 0 0" />
<parent
        link="link_2" />
<child
        link="link_4" />
<axis
        xyz= "0 0 -1"/>
<limit
        lower="-0.7"
        upper="0.7"
        effort="50"
        velocity="1" />
<dynamics
        friction="1" />
</joint>

<link
    name="link_5">
    <inertial>
        <origin
            \(x y z=" 0.112038450872787-3.68252928151236 \mathrm{E}-110.000372989461207632 "\)
            rpy="0 0 0" />
        <mass
            value="0.0272492455206807" />
        <inertia
            ixx="6.82954783932986E-07"
            ixy="2.4502489465766E-15"
            ixz="8.26095767314848E-08"
            iyy="0.000108754287152569"
            iyz="-5.71025720952082E-16"
            izz="0.000108813944757045" />
        </inertial>
    <visual>
        <origin
            xyz="0 0 0"
            rpy="0 0 0" />
        <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_5.STL" />
        </geometry>
        <material
            name=" ">
            <color
                rgba="0.0941176470588235 0.0941176470588235 0.0941176470588235 1" />
        </material>
    </visual>
    <collision>
        <origin
            \(x y z=" 000 "\)
    ```
            rpy="0 0 0" />
        <geometry>
            <mesh
            filename="package://robotic_leg/stl_files/link_5.STL" />
        </geometry>
    </collision>
</link>
<joint
    name="joint_A"
    type="revolute">
    <origin
        xyz="0.010032 0.084403 0.014867"
        rpy="0 -0.0016667 0.0011098" />
    <parent
        link="link_4" />
    <child
        link="link_5" />
    <axis
        xyz="0 0 -1" />
    <limit
        lower="-3.14"
        upper="3.14"
        effort="50"
        velocity="1" />
    <dynamics
        friction="1" />
    </joint>
<!-- Transmissions for ROS Control -->
<transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint_0">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<transmission name="tran2">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint_1">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor2">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<transmission name="tran3">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint_2">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
```

```
</joint>
<actuator name="motor3">
    <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<gazebo>
    <plugin filename="libgazebo_ros_control.so" name="gazebo_ros_control">
    <robotNamespace>/leg_controller</robotNamespace>
    </plugin>
</gazebo>
<gazebo>
    <joint name="fake_joint" type="fixed">
        <pose>0 0 0 0 1.5708 0</pose>
        <parent>link_3B</parent>
        <child>link_5</child>
    </joint>
</gazebo>
</robot>
```


## A. 2 MATLAB Code

In the following the MATLAB code for generating the leg's trajectory is reported.

```
%load mocap data from file
data = load('dog_data.mat').data;
%set kinematic parameters
lla = 0.07875;
l1b = 0.11;
l2 = 0.3;
l3 = 0.3;
time = data(7:560 , 2)*20;
X_p = data(7:560 , 36).';
Y_p = data(7:560 , 37).';
Z_p = data(7:560 , 38).';
```

\%Now take 3 points on the dog body
X_LFW = data(7:560 , 60);
Y_LFW = data(7:560, 61);
Z_LFW = data(7:560, 62);
X_RFW = data(7:560 , 63);
Y_RFW = data(7:560 , 64);
Z_RFW = data(7:560, 65);
\%This will be the origin of our RF
X_LBW = data(7:560 , 66)
Y_LBW = data(7:560 , 67) ;
Z_LBW = data(7:560 , 68);
\%plot the trajectory seen from the camera reference frame
figure;
plot(Y_LBW/1000 + 4, Z_LBW*4/1000 - 1.4 , 'LineWidth',1); xlim([0 , 8]);
xlabel('X direction[m]'); ylabel('Z direction[m]');
title('Dog toe trajectory w.r.t. camera reference frame');
T = [X_LBW , Y_LBW , Z_LBW].'; \%series of translation vectors
a = [X_LFW , Y_LFW , Z_LFW];
b = [X_LBW , Y_LBW , Z_LBW];
c = [X_RFW , Y_RFW , Z_RFW];
$x=(a-b) . ' ;$
y_raw = (c - a).';
z = [];
y = [];
point_traj = [];
for $i=1: 554$
\%generate a vector which will be for sure normal to x
temp $=$ cross(x(:,i) , y_raw(:,i));
temp = temp / norm(temp)
z = [z , temp];
\%now generate the new $y$ normal to $x$ and $z$

ETSEIB

```
    y_temp = cross(z(:,i),x(:,i));
    y_temp = y_temp / norm(y_temp);
    y = [y , y_temp];
    %rotation matrix from camera RF to robot RF
    R = [x(:,i)/norm(x(:,i)) , y_temp , temp];
    %homogeneous transformation matrix
    M_temp = [R , T(:,i);
        0 0 0 1];
    M = inv(M_temp);
    point_traj = [point_traj , M * ([X_p(i) ; Y_p(i) ; Z_p(i) ; 1])-[0.7 0 0 0].'];
end
var_x.time = time(400:490) - 133;
var_x.signals(1).values = flipud(point_traj(1,400:490).' / 2000 - 0.2);
var_z.time = time(400:490) - 133;
var_z.signals(1).values = flipud(-point_traj(3,400:490).' / 1000 + 0.0);
%now open the Simulink model for the experimental setup
open('IJ_right_model_REAL.slx');
sim('IJ_right_model_REAL.slx');
%save data from experimental test
save('my_data.mat' , 'perro')
%Execute the Gazebo simulation
open('IJ_right_model.slx');
sim('IJ_right_model.slx');
%save data from Gazebo simulation
save('data_gazebo.mat' , 'perro_gazebo')
%Execute the Simulink simulation
open('Inverse_Jacobian.slx');
sim('Inverse_Jacobian.slx');
%save data from simulation
save('data_simulink.mat' , 'perro_simulink')
```

In the following the MATLAB code for plotting the simulation and experimental data is reported.

```
%load signals from experimental setup
load('my_data.mat');
%plot the followed trajectory and the desired one
figure(1);
plot(perro.signals(4).values(:,1).' , perro.signals(4).values(:,3).' , 'color' , 'blue')
    ;
hold on;
plot(perro.signals(5).values(:,1).' , perro.signals(5).values(:,3).' , 'color' , 'red');
grid on;
legend({'Desired trajectory','Followed trajectory'});
xlabel('X_R axis [m]'); ylabel('Z_R axis [m]');
title('Comparison between desired and actual trajectory in experimental setup');
%compute now the norm of the error between x_d and x_e
error_norm_exp = [];
for i=1:height(perro.time)
    n_temp_exp = perro.signals(3).values(i,1)^2 + perro.signals(3).values(i,2)^2 + perro.
        signals(3).values(i,3)^2;
    n_temp_exp = sqrt(n_temp_exp);
    error_norm_exp = [error_norm_exp , n_temp_exp];
end
%plot now the norm of the error between x_d and x_e
figure(2);
plot(perro.time.' , error_norm_exp , 'color' , 'blue');
hold on;
grid on;
xlabel('Time [s]'); ylabel('Error x_e - x_d [m]');
title('Absolute error x_e - x_d [m] in operational space, experimental results');
%plot now the joints positions
figure(3);
subplot(2,1,1);
plot(perro.time.' , perro.signals(1).values(:,1).' , 'color' , 'blue');
hold on;
plot(perro.time.' , perro.signals(1).values(:,2).' , 'color' , 'red');
hold on;
plot(perro.time.' , perro.signals(1).values(:,3).' , 'color' , 'green');
hold on;
grid on;
legend({'q-1','q-2','q_3'});
xlabel('Time [s]'); ylabel('Joint position [rad]');
title('Joint positions, experimental results');
%plot now the joints velocities
%figure(4);
subplot(2,1,2);
plot(perro.time.' , perro.signals(2).values(:,1).' , 'color' , 'blue');
hold on;
plot(perro.time.' , perro.signals(2).values(:,2).' , 'color' , 'red');
hold on;
```

```
plot(perro.time.' , perro.signals(2).values(:,3).' , 'color' , 'green');
hold on;
grid on;
legend({'q-1','q_2','q-3'});
xlabel('Time [s]'); ylabel('Joint velocity [rad/s]');
title('Joint velocities, experimental results');
%%
%Load signals from Gazebo simulation
load('data_gazebo.mat');
%plot the followed trajectory and the desired one
figure(5);
plot(perro_gazebo.signals(4).values(:,1).' , perro_gazebo.signals(4).values(:,3).' ,
    color' , 'blue');
hold on;
plot(perro_gazebo.signals(5).values(:,1).' , perro_gazebo.signals(5).values(:,3).' ,
    color' , 'red');
grid on;
legend({'Desired trajectory','Followed trajectory'});
xlabel('X_R axis [m]'); ylabel('Z_R axis [m]');
title('Comparison between desired and actual trajectory in Gazebo simulation');
%compute the norm of the error between x_d and x_e
error_norm = [];
for i=1:height(perro_gazebo.time)
    n_temp = perro_gazebo.signals(3).values(i,1)^2 + perro_gazebo.signals(3).values(i,2)
        ^2 + perro_gazebo.signals(3).values(i,3)^2;
    n_temp = sqrt(n_temp);
    error_norm = [error_norm , n_temp];
end
%plot the norm of the error between x_d and x_e
figure(6);
plot(perro_gazebo.time.' , error_norm , 'color' , 'blue');
hold on;
grid on;
xlabel('Time [s]'); ylabel('Error x_e - x_d [m]');
title('Absolute error x_e - x_d [m] in operational space, Gazebo simulation');
%plot now the joints positions
figure(7);
subplot(2,1,1);
plot(perro_gazebo.time.' , perro_gazebo.signals(1).values(:,1).' , 'color' , 'blue');
hold on;
plot(perro_gazebo.time.' , perro_gazebo.signals(1).values(:,2).' , 'color' , 'red');
hold on;
plot(perro_gazebo.time.' , perro_gazebo.signals(1).values(:,3).' , 'color' , 'green');
hold on;
grid on;
legend({'q_1','q_2','q_3'});
xlabel('Time [s]'); ylabel('Joint position [rad]');
title('Joint positions, Gazebo simulation');
%plot now the joints velocities
```

```
subplot(2,1,2);
plot(perro_gazebo.time.' , perro_gazebo.signals(2).values(:,1).' , 'color' , 'blue');
hold on;
plot(perro_gazebo.time.' , perro_gazebo.signals(2).values(:,2).' , 'color' , 'red');
hold on;
plot(perro_gazebo.time.' , perro_gazebo.signals(2).values(:,3).' , 'color' , 'green');
hold on;
grid on;
legend({'q_1','q_2','q_3'});
xlabel('Time [s]'); ylabel('Joint velocity [rad/s]');
title('Joint velocities, Gazebo simulation');
%%
%Load signals from simulation results
load('data_simulink.mat');
%plot the followed trajectory and the desired one
figure(9);
plot(perro_simulink.signals(4).values(:,1).' , perro_simulink.signals(4).values(:,3).' ,
    'color' , 'blue');
hold on;
plot(perro_simulink.signals(5).values(:,1).' , perro_simulink.signals(5).values(:,3).' ,
    'color' , 'red');
grid on;
legend({'Desired trajectory','Followed trajectory'});
xlabel('X_R axis [m]'); ylabel('Z_R axis [m]');
title('Comparison between desired and actual trajectory in simulation results');
%compute now the error between x_d and x_e
error_norm_sim = [];
for i=1:height(perro_simulink.time)
    n_temp_sim = perro_simulink.signals(3).values(i,1)^2 + perro_simulink.signals(3).
        values(i,2)^2 + perro_simulink.signals(3).values(i,3)^2;
    n_temp_sim = sqrt(n_temp_sim);
    error_norm_sim = [error_norm_sim , n_temp_sim];
end
%plot now the error between x_d and x_e
figure(10);
plot(perro_simulink.time.' , error_norm_sim, 'color' , 'blue');
hold on;
grid on;
xlabel('Time [s]'); ylabel('Error x_e - x_d [m]');
title('Absolute error x_e - x_d [m] in operational space, simulation results');
%plot now the joints positions
figure(11);
subplot(2,1,1);
plot(perro_simulink.time.' , perro_simulink.signals(1).values(:,1).' , 'color' , 'blue')
    ;
hold on;
plot(perro_simulink.time.' , perro_simulink.signals(1).values(:,2).' , 'color' , 'red');
hold on;
plot(perro_simulink.time.' , perro_simulink.signals(1).values(:,3).' , 'color' , 'green'
    );
```

```
hold on;
grid on;
legend({'q_1','q_2','q_3'});
xlabel('Time [s]'); ylabel('Joint position [rad]');
title('Joint positions, simulation results');
%plot now the joints velocities
subplot(2,1,2);
plot(perro_simulink.time.' , perro_simulink.signals(2).values(:,1).' , 'color' , 'blue')
    ;
hold on;
plot(perro_simulink.time.' , perro_simulink.signals(2).values(:,2).' , 'color' , 'red');
hold on;
plot(perro_simulink.time.' , perro_simulink.signals(2).values(:,3).' , 'color' , 'green'
    );
hold on;
grid on;
legend({'q_1','q_2','q_3'});
xlabel('Time [s]'); ylabel('Joint velocity [rad/s]');
title('Joint velocities, simulation results');
```

In the following the MATLAB code for computing the forward and inverse kinematics is reported.

```
%%MATLAB script to solve the Forward and Inverse kinemtics problems
syms q3 q1 q2 lla llb l2 l3 px py pz
%set the DH matrices
AR = [0 0 - 1 0; 1 0 0 0; 0 -1 0 0; 0 0 0 1];
A1 = [cos(q1) -sin(q1) 0 0;
    sin(q1) cos(q1) 0 0;
    0 0 1 l1b;
    0 0 0 1] * [0 0 1 0;
                    10 0 0;
                0 1 0 0;
                0 0 0 1];
A2 = [cos(q2) - sin(q2) 0 0;
    sin(q2) cos(q2) 0 0;
    0 0 1 lla;
    0 0 0 1] * [1 0 0 l2;
            0-1 0 0;
            0 0-1 0;
            0 0 0 1];
A3 = [cos(q3+pi/2) - sin(q3+pi/2) 0 0
    sin(q3+pi/2) cos(q3+pi/2) 0 0
    0 0 1 0;
    0 0 0 1]* [1 0 0 l3;
            0 1 0 0;
            0 0 1 0;
            0 0 0 1];
%compute the homogeneous transformation from robot RF to point P
T_tot = simplify(AR * A1 * A2 * A3)
%determine the forward kinematics equations
pe = simplify(AR * A1 * A2 * A3 * [0;0;0;1])
%Compute Analytic Jacobian
J_A = simplify(jacobian([pe(1),pe(2),pe(3)] , [q1,q2,q3]))
%Solve the inverse kinematics equations
B = simplify(inv(A1) * inv(AR) * [px;py;pz;1])
C = simplify(A2 * A3 * [0;0;0;1])
eq1 = B(1) == C(1)
eq2 = B(2) == C(2)
eq3 = B(3) == C(3)
S = solve([eq1 , eq2 , eq3] , [q1,q2,q3])
```


## A. 3 C++ Code

In the following the $\mathrm{C}++$ code relative to the ROS node is reported.

```
#include <ros/ros.h>
#include <trajectory_msgs/JointTrajectory.h>
#include <trajectory_msgs/JointTrajectoryPoint.h>
#include <control_msgs/FollowJointTrajectoryAction.h>
#include <actionlib/client/simple_action_client.h>
#include <geometry_msgs/Pose.h>
#include <std_msgs/Float64MultiArray.h>
#define DEG_2_RAD 3.14/180
#define NUM_JOINTS 3
//create an array to contain the joint positions
std::array<double,NUM_JOINTS> positions;
bool moveRobot(std::array<double,NUM_JOINTS> &conf, double moveduration,
        actionlib::SimpleActionClient<control_msgs::FollowJointTrajectoryAction> &rClient)
{
    control_msgs::FollowJointTrajectoryGoal goal;
    goal.trajectory.header.stamp = ros::Time::now()+ros::Duration(0.001);
    //set active joints names
    goal.trajectory.joint_names.resize(NUM_JOINTS);
    goal.trajectory.joint_names[0] = "joint_0";
    goal.trajectory.joint_names[1] = "joint_1";
    goal.trajectory.joint_names[2] = "joint_2";
    //set the right joint positions to be sent
    goal.trajectory.points.resize(1);
    goal.trajectory.points[0].positions.resize(NUM_JOINTS);
    goal.trajectory.points[0].positions[0] = conf[0];
    goal.trajectory.points[0].positions[1] = conf[1];
    goal.trajectory.points[0].positions[2] = conf[2];
    goal.trajectory.points[0].time_from_start = ros::Duration(moveduration);
    //Send goal
    rClient.sendGoal(goal);
    //Wait for the action to return
    bool finished_before_timeout = rClient.waitForResult(
        goal.trajectory.points.back().time_from_start+ros::Duration(2*moveduration));
    actionlib::SimpleClientGoalState state = rClient.getState();
    if (finished_before_timeout)
    {
        ROS_INFO("Robot action finished: %s",state.toString().c_str());
    }
    else
    {
        ROS_ERROR("Robot action did not finish before the timeout: %s",
        state.toString().c_str());
```

```
    }
    return (state == actionlib::SimpleClientGoalState::SUCCEEDED);
}
void MATLABcmd(const geometry_msgs::Pose& msg)
{
    /*
        read the topic published by Simulink and fill the
        joint position vector with the corresponding information
    */
    positions[0] = msg.position.x;
    positions[1] = msg.position.y;
    positions[2] = msg.position.z;
}
int main(int argc, char **argv)
{
    //initialize the ROS system and become a node.
    ros::init(argc, argv, "single_joint_control");
    ros::NodeHandle nh;
    //create a subscriber object to receive topics from Simulink
    ros::Subscriber sub = nh.subscribe("joint_state_publisher", 1000, &MATLABcmd);
    //create the controller action
    actionlib::SimpleActionClient<control_msgs::FollowJointTrajectoryAction> robotClient(
            "/leg_controller/joint_traj_controller_parallel/follow_joint_trajectory");
    if(!robotClient.waitForServer(ros::Duration(5.0)))
    {
        ROS_ERROR("action server not available");
    };
    //set rate of the ROS node to 1kHz
    double cycletime = 0.001;
    ros::Rate rate(l/cycletime);
    //init the vector of joint positions
    positions[0] = 0.0;
    positions[1] = 0.0;
    positions[2] = 0.0;
    while(ros::ok())
    {
        //execute the pending callbacks
        ros::spinOnce();
        //execute the action to send trajectory command
        moveRobot(positions, cycletime, robotClient);
        rate.sleep();
    }
}
```


## A. 4 ROS launch files

In the following the gazebo_parallel.launch file for spawning the 3D model of the leg in Gazebo is reported.

```
<launch>
    <include
        file="$(find gazebo_ros)/launch/empty_world.launch"
    />
    <node
        name="spawn_model"
        pkg="gazebo_ros"
        type="spawn_model"
        args="-file $(find robotic_leg)/urdf/leg_new_links.urdf -urdf -model leg_new_links"
        output="screen"
    />
</launch>
```

In the following the leg_server_parallel.launch file for launching the necessary ROS nodes is reported.

```
<?xml version="1.0"?>
<launch>
    <param name="robot_description"
        textfile="$(find robotic_leg)/urdf/leg_new_links.urdf" />
    <!-- Include the launch file for spawning the robotic leg in gazebo -->
    <include
        file="$(find robotic_leg)/launch/gazebo_parallel.launch"
    />
    <node
        name="joint_control_node"
        pkg="robotic_leg"
        type="key_joint_control_parallel"
        launch-prefix="xterm -e"
    />
    <!-- load the controllers -->
    <rosparam file="$(find robotic_leg)/controller/leg_controller.yaml"
            command="load"/>
    <!-- load the parameters -->
    <rosparam file="$(find robotic_leg)/config/gazebo_ros_control_params_parallel.yaml"
                command="load"/>
    <!-- spawn the controllers -->
    <node
        name="joint_traj_controller_spawner"
        pkg="controller_manager"
        type="spawner"
        respawn="true"
        launch-prefix="xterm -e"
        ns="/leg_controller"
```

```
    args="joint_traj_controller_parallel joint_state_controller"
    />
</launch>
```

