Robotics Lab Final Project: Design of a Navigation and Control Framework for a Quadruped Robot

T. Rebeggiani, P38000036

April 6, 2022

Abstract

Quadruped Robots are mobile robotic platforms of increasing relevance due to their ability to navigate through uneven, rough terrain and perform complex tasks such as stair-climbing. This is why they are becoming increasingly popular in practical applications such as surveillance or automated plant inspection, among others. In this work, I present the practical steps towards the implementation of a control and navigation framework for a React Robotics DogBot Quadruped Robot. The coding is done in C++, using ROS middleware and Gazebo as a simulation platform. The report mostly focuses on software development, and is to be intended as a tutorial to build and test control algorithms for complex robotic systems using ROS and Gazebo. The code related to the implementation of the navigation and control framework is available on Github.

Contents

Supplementary Material

The Github repository related to the implementation of the navigaton and control framework described in this work is available at `https://github.com/trebeggiani/rl-project`

Introduction

While still in the early stages of commercial deployment, legged robots have already shown their disruptive potential in areas such as construction, defense, and exploration. They are bound to lead the next generation of mobile robots.

The main challenges that are being tackled today by the legged robots community involve gait planning, dealing with disturbances, and state estimation algorithms, all of which become relevant problems when robots need to navigate in cluttered environments. In fact, one may argue that the key edge that is offered by quadruped robots, and at the same time the most difficult aspect to deal with in the development stages, is robustness. Some of these challenges above are being addressed with the use of machine learning and simulation-based training, as in [5]. The proponents of learning in legged robotics often argue that this is the only way that robustness can be guaranteed in such complex systems.

However, for now, classical optimal control is the de-facto standard for legged robots (see, for example, [3] and [2]). Namely, a quadratic programming (QP) problem is often set, which aims at following a reference for the body frame of the robot. This whole-body QP control allows to conveniently include the robot's model, stance feet contact, joint and torque limits, swing feet references, and the estimation of disturbances as constraints.

The design of the low-level controller is often based on the center of mass (CoM) model of the quadruped robot, in view of the relevant simplifications it entails in the synthesis process. The whole-body QP controller presented here is taken from [10]. there is no contribution brought forward by this work in this regard.

As regards the navigation problem, that is addressed here using an online version of the artificial potential fields planning algorithm, a very simple yet effective algorithm developed first by Khatib in [6].

The control and navigation algorithms have been tested on the DogBot quadruped robot (Fig. 1), a robotic platform developed by React Robotics. The URDF model of the robot is publicly available on Github [1].

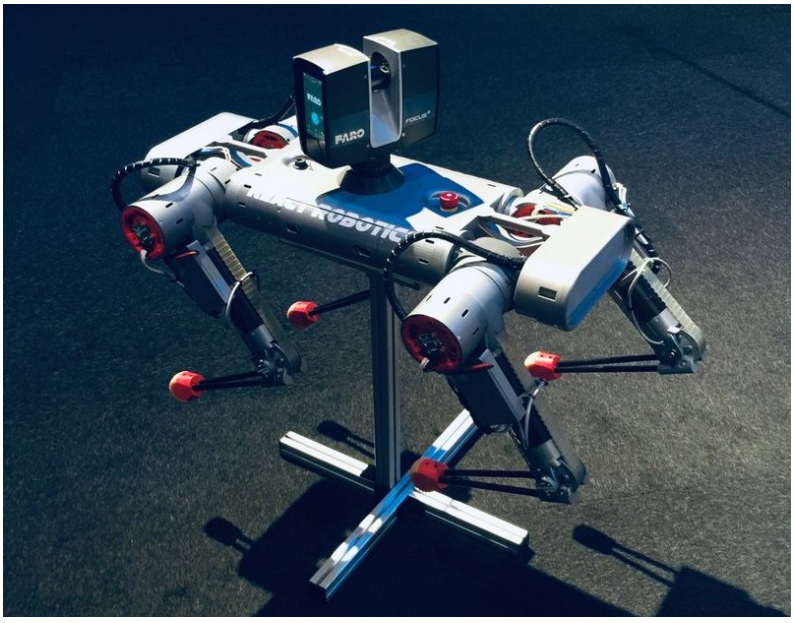

Figure 1: A React Robotics DogBot quadruped robot. This model was fitted with a Faro 3D laser scanner. Such a configuration is highly useful in applications such as mapping of construction sites.

The task that is assigned to the robot and showcased in the attached demonstration video (which you can find in the `media` folder) consists in navigating a small-scale

[1] GitHub repository of the React Robotics Dogbot quadruped robot `https://github.com/ReactRobotics/DogBotV4`

environment composed of three rooms, three of whom contain QR codes that must be scanned by the robot until it finds the one corresponding to the Id specified by the user. The coordinates of each QR code's location are known, and the robot must head towards them by means of an online version of the artificial potential fields method. A view of the robot's workspace can be found in Figure 6.

The structure of this work is composed as follows:

Section 1 deals with the CoM-based modeling of a generic quadruped robotic platform.

Section 2 describes in detail the full QP-based architecture used in the low-level control of the quadruped robot, including the momentum-based disturbance estimator that was mentioned before.

Section 3 presents the artificial potential fields planning algorithm.

Section 4, which is to be intended as the main contribution of this work, illustrates the coding process that was followed during the development stage of the project, giving a comprehensive walkthrough of the software that is found in the enclosed repository.

1 Modeling

Quadruped robots are floating base systems. Such systems are not constrained in any way to the reference (world) frame, and thus their state must be described taking into account the coordinates of the body frame w.r.t. an external reference frame. In our case, this means that the full set of state variables is made of 12 joint variables plus 6 variables for a minimal representation of the position and orientation of the body frame. As stated before, a convenient way to describe a floating base legged robot is to express its dynamics considering the CoM instead of a generic body frame. This way, the structure of the model is greatly simplified, as its dynamics become

$$M(q)\dot{v} + h(q,v) = S^T\tau + J_{st}(q)^T f_{gr} + J(q)^T f_{ext}, \tag{1}$$

where

$$v = \begin{bmatrix} \dot{x}_{\text{com}}^T & \omega_{\text{com}}^T & \dot{q}^T \end{bmatrix}^T, \tag{2}$$

$$M(q) = \begin{bmatrix} M_{com}(q) & O_{6\times nnl} \\ O_{nn_x\times 6} & M_q(q) \end{bmatrix} \in \mathbb{R}^{6+nn_l\times 6+nn_l}, \tag{3}$$

$$h(q,v) = \begin{bmatrix} O_{6\times(6+nn_l)} \\ C_q(q,v) \end{bmatrix} v + \begin{bmatrix} mg \\ 0_{nn_l} \end{bmatrix}. \tag{4}$$

As can be seen from the structure of $M(q)$, there is a clear decoupling between the CoM part and the joints part of the equations, that brings the advantage of easing the computational load of the controller. Further, $C_q(q,v) \in \mathbb{R}^{nn_l\times(6+nn_l)}$, $g = \begin{bmatrix} g_0^T & 0_3^T \end{bmatrix}^T \in \mathbb{R}^6$ and $S = \begin{bmatrix} O_{nn_l\times 6} & I_{nn_l} \end{bmatrix}$. The Jacobian $J(q) = [J_{com}(q) \quad J_j(q)] \in \mathbb{R}^{3n_l\times 6+nn_t}$ maps the velocities of the CoM and of all the leg joints onto feet velocities, whereas $J_{st}(q) = [J_{st,com}(q) \quad J_{st,j}(q)] \in \mathbb{R}^{3n_{st}\times 6+nn_l}$ only considers the swing legs. Therefore, the transpose matrices of these Jacobians map forces acting at the feet onto forces at the CoM and joint torques.

2 Low-level Control

This section deals with the implementation of the low-level control algorithm. As was stated before, this work brings no contributions in this aspect, as the architecture presented here is the one proposed in [10]. While this report focuses on the practical design of the control and navigation algorithm, a brief overview of the controller's components is still given here. The structure of the controller can be found in Fig. 2, taken from [10]. The controller is mainly composed of two blocks: one that deals with planning a reference trajectory, and one that computes the optimal torques solving a quadratic programming problem.

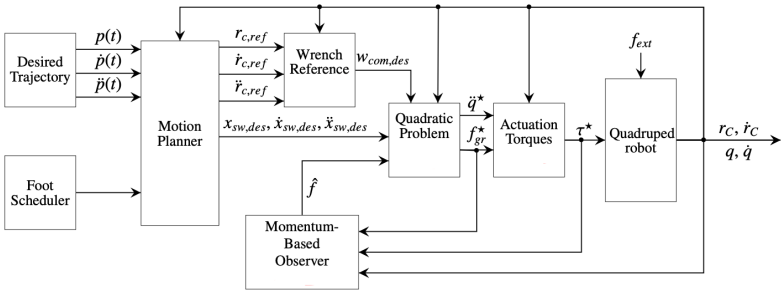

Figure 2: Low-level controller scheme.

2.1 Motion Planner

The role of the motion planner is that of taking the desired trajectory for CoM of the robot and computing a reference for the CoM as well as for the swing legs, in order to keep the robot dynamically balanced during its dynamic phase. The motion planner also takes the foot schedule as an input, that is the stance/swing sequence for the four legs.

The motion planner interpolates the points of the desired trajectory with a spline, solving a nonlinear programming problem that is constrained by the zero-moment point (ZMP) criterion[2]. Namely, the ZMP must always lie inside the support polygon of the robot to keep the robot from falling [11]. To compute the swing leg reference, the motion planner uses two more splines per swing foot (one to lift the foot and one to lower it). Further details about the motion planner will be omitted for the sake of simplicity. To implement the motion planner in ROS, the Towr library was used [12].

2.2 QP-based Whole-body Controller

Before getting to the ROS implementation, let us first start with a (brief) theoretical introduction to the controller scheme that will be adopted. As was stated before, most control strategies for legged robots use a QP-base whole-body controller in view of its intuitive nature and generally robust performance (e.g. in [3, 2]). In our case, the controller solves a quadratic optimization problem which can be written as

$$\begin{aligned} \min_{\zeta} \quad & f(\zeta) \\ \text{s.t.} \quad & A\zeta = b \\ & D\zeta \leq c \end{aligned} \quad , \tag{5}$$

with

$$f(\zeta) = \left\| \bar{J}_{st,c}^T \Sigma \zeta + \bar{J}_{st,c}^T \hat{f}_{st} - W_{CoM,des} \right\| + \|\zeta\|, \tag{6}$$

$$\zeta = \begin{bmatrix} \ddot{r}_c^T & \ddot{q}_j^T & f_{gr}^T \end{bmatrix}^T \in \mathbb{R}^{n_b+n_j+3n_{st}}, \tag{7}$$

$$r_c = \begin{bmatrix} p_c^T & \eta_c^T \end{bmatrix}^T. \tag{8}$$

In (6), $\Sigma \in \mathbb{R}^{n_b+n_j+3n_{st}}$ is a matrix that selects the last $3n_{st}$ elements of ζ (i.e. the ground reaction forces). Notice that only the first part of the objective function aims at tracking the wrench reference, whereas the second part is needed to keep joint and CoM accelerations down as well. The actual wrench acting on the CoM depends on both the state variables and the disturbances acting at the stance feet.

To estimate the disturbance at the feet that is needed to set the cost function at each controller iteration, the authors of [10] use a momentum-based observer. This observer reconstructs disturbing forces resulting from both unmodeled dynamics and

[2]The ZMP, also known as the center of pressure (CoP), is the projection of the CoM coordinates onto the xy-plane.

external forces such as pushing actions or collisions with obstacles. This momentum-based observed, however, is not used here. This means that the cost function term $\bar{J}_{st,c}^T \hat{f}_{st}$ is always considered null.

To compute the cost function, we also need the wrench reference $W_{CoM,des}$, which can be assigned via a PDD control policy with gravity compensation, as

$$W_{CoM,des} = -K_p \left(r_c - r_{c,ref}\right) - K_d \left(\dot{r}_c - \dot{r}_{c,ref}\right) + mg + M_{CoM}(q)\ddot{r}_{c,ref}. \quad (9)$$

The QP controller also needs constraints. They impose the requirements of dynamic consistency (control actions must be consistent with the system's dynamics), non-sliding ground-feet contact, compliance with joint actuators torque limits and trajectory following for the swing feet. Merging together all equations and inequalities, we obtain the following matrices:

$$A = \begin{bmatrix} M_{CoM}(q) & O_{n_b \times n_j} & -\bar{J}_{st,c}^T & O_{n_b \times 3n_{sw}} \\ \bar{J}_{st,c} & \bar{J}_{st,j} & O_{3n_{st} \times 3n_{st}} & O_{3n_{st} \times 3n_{sw}} \end{bmatrix}, \quad (10)$$

$$b = \begin{bmatrix} -mg \\ -\dot{\bar{J}}_{st,c}\dot{r}_c - \dot{\bar{J}}_{st,j}\dot{q}_j \end{bmatrix}, \quad (11)$$

$$D = \begin{bmatrix} 0_{4n_{st}} \times 6 & O_{4n_{st} \times n_j} & D_{fr} & O_{4n_{st} \times 3n_{sw}} \\ O_{n_j \times n_b} & M_{22} & -\bar{J}_{st,j}^T & O_{n_j \times 3n_{sw}} \\ O_{n_j \times n_b} & -M_{22} & \tilde{J}_{st,j}^T & O_{n_j \times 3n_{sw}} \\ \bar{J}_{sw,c} & \bar{J}_{sw,j} & 0_{3n_{sw} \times 3n_{st}} & I_{3n_{sw} \times 3n_{sw}} \\ -\bar{J}_{sw,c} & -\bar{J}_{sw,j} & 0_{3n_{sw} \times 3n_{st}} & I_{3n_{sw} \times 3n_{sw}} \end{bmatrix}, \quad (12)$$

$$c = \begin{bmatrix} 0_{4n_{st}} \\ \tau_{\max} - \bar{C}_2 \dot{q}_j \\ -\left(\tau_{\min} - \bar{C}_2 \dot{q}_j\right) \\ \ddot{x}_{sw,cmd} - \dot{\bar{J}}_{sw,c}\dot{r}_c - \dot{\bar{J}}_{sw,j}\dot{q}_j \\ -\ddot{x}_{sw,cmd} + \dot{\bar{J}}_{sw,c}\dot{r}_c + \dot{\bar{J}}_{sw,j}\dot{q}_j \end{bmatrix}. \quad (13)$$

To get more details about the elements of matrices A, b, D, c, please see [10]. After the quadratic problem is solved at a given time step, we are ready to feed the reference to the torque-controlled actuators, computing it as

$$\tau^* = M_q(q)\ddot{q} + C_q(q,v)\dot{q} - J_{st,j}(q)^T f_{gr^*} \quad (14)$$

Now that the structure of the controller is clear, we can turn to the higher level part of the pipeline. the implementation of all software modules is presented in section 4.

3 Navigation

As we can see in Fig. 2, the planner (in our case, Towr) needs some kind of reference for the robot body frame. It is possible to assign this reference using one of the numerous planning algorithms which have proved their value during the now quite long history of mobile robots.

Despite the robot's base having 6 degrees of freedom (DoFs), we do not wish to control all of them. Instead, a 2D planner can be used, to control only the X-Y position of the base, aloong with the yaw. The remaining DoFs are assigned according to the nominal pose of the robot. In other terms, the base reference is always fixed at $z_r = 0.43m$, whereas the roll and pitch angle references are always set to zero.

Given the complexity of more advanced sampling-based algorithms such as RRT* or PRM, the project task sheet requires, as a specification, the use of the artificial potential fields planning algorithm. This planner, proposed by Khatib in 1986 [6], is a quite old algorithm with many drawbacks. The main drawback consists in the problem of *local minima* (Fig. 3), which can be dealt with in different ways. However, a plus offered by the artificial potential fields method is its simplicity and real-time

capability. In fact, the implementation of the algorithm's core requires only a handful of C-style lines of code.

The key idea on which the artificial potential fields method is based is that of (artificially) constructing an attractive potential to guide the robot towards the goal. This *attractive* potential—in our case a conical function (15)—is countered by a *repulsive* potential, which is designed to keep the robot away from the *C-obstacle* region, the occupied part of the configuration space.

Since the occupied space cannot usually be represented as a single point, a different approach must be taken to compute the repulsive potential from a known map of the environment. We can assign it for each i-th convex component of the obstacle region as in (16), where $k_{r,i}$ is a parameter to be tuned, $\eta_i(\boldsymbol{q})$ is the distance of the configuration $\boldsymbol{q}$ from the i-th convex component and $\eta_{0,i}$ is the *range of influence* of the obstacle. Inside the range of influence, the repulsive potential tends to infinity as the distance nears zero. The distance computation is arguably the only computationally expensive part of the potential fields method, as we need to query each occupied cell of the occupancy grid at every iteration.

As a side note, the artificial potential algorithm can also be used offline, if a map is already present. This speeds things up considerably. However, the online version of the algorithm can be used in exploration tasks such as the one that is assigned here, where a map of the environment is not given, and must be constructed on-the-fly with a SLAM algorithm.

$$U_a(\boldsymbol{q}) = k_a \|\boldsymbol{e}(\boldsymbol{q})\| \tag{15}$$

$$U_{r,i}(\boldsymbol{q}) = \begin{cases} \frac{k_{r,i}}{\gamma}\left(\frac{1}{\eta_i(\boldsymbol{q})} - \frac{1}{\eta_{0,i}}\right)^{\gamma} & \text{if } \eta_i(\boldsymbol{q}) \leq \eta_{0,i} \\ 0 & \text{if } \eta_i(\boldsymbol{q}) > \eta_{0,i} \end{cases} \tag{16}$$

$$U_t(\boldsymbol{q}) = U_a(\boldsymbol{q}) + U_r(\boldsymbol{q}) \tag{17}$$

To be able to compute viable references for Towr, we must first compute the negative gradient of these potentials, obtaining artificial attractive and repulsive forces. The attractive force (18) points towards the goal, and its norm decreases as we approach the goal. The repulsive force (19) points away from the nearest occupied grid cell, and its norm increases as we get closer to obstacles.

$$\boldsymbol{f}_{a2}(\boldsymbol{q}) = -\nabla U_{a2}(\boldsymbol{q}) = k_a \frac{\boldsymbol{e}(\boldsymbol{q})}{\|\boldsymbol{e}(\boldsymbol{q})\|} \tag{18}$$

$$\boldsymbol{f}_{r,i}(\boldsymbol{q}) = -\nabla U_{r,i}(\boldsymbol{q}) = \begin{cases} \frac{k_{r,i}}{\eta_i^2(\boldsymbol{q})}\left(\frac{1}{\eta_i(\boldsymbol{q})} - \frac{1}{\eta_{0,i}}\right)^{\gamma-1} \nabla\eta_i(\boldsymbol{q}) & \text{if } \eta_i(\boldsymbol{q}) \leq \eta_{0,i} \\ 0 & \text{if } \eta_i(\boldsymbol{q}) > \eta_{0,i} \end{cases} \tag{19}$$

$$\boldsymbol{f}_t(\boldsymbol{q}) = -\nabla U_t(\boldsymbol{q}) = \boldsymbol{f}_a(\boldsymbol{q}) + \sum_{i=1}^{p} \boldsymbol{f}_{r,i}(\boldsymbol{q}) \tag{20}$$

These "forces" can be conveniently summed together, and used as acceleration or velocity references when we deal with simple robotic platforms. However, our low-level planner, Towr, needs positon references. A quick work-around consists in multiplying force values by the time step of the algorithm loop (or, equivalently, an arbitrary small value) and to add the result to the current position. This way, we obtain waypoints that can be sent to Towr. Further, the resulting position offset should be truncated if over a certain value, to avoid exceeding mechanical constraints imposed by the chosen Towr gait. In fact, Towr tries to reach each waypoint in a given number of steps. If the offset is too big, the optimizer may not find a solution to the problem, or may output de-stabilizing references. A caveat that was previously mentioned consists in the problem of local minima. When we sum up the attractive and repulsive forces as in (20), local minima, different from the goal configuration, may arise. This can lead to the robot being trapped inside confined regions of the configuration space

(see Fig. 3). A workaround to solve this problem consists in bypassing the algorithm whenever a stall situation is detected, and compute a heuristic reference to exit the basin of attraction of the minimum. However, particularly critical situations, such as a task that consists in navigating between two adjacent rooms separated by a long wall, cannot be handled by the planner in an online setting.

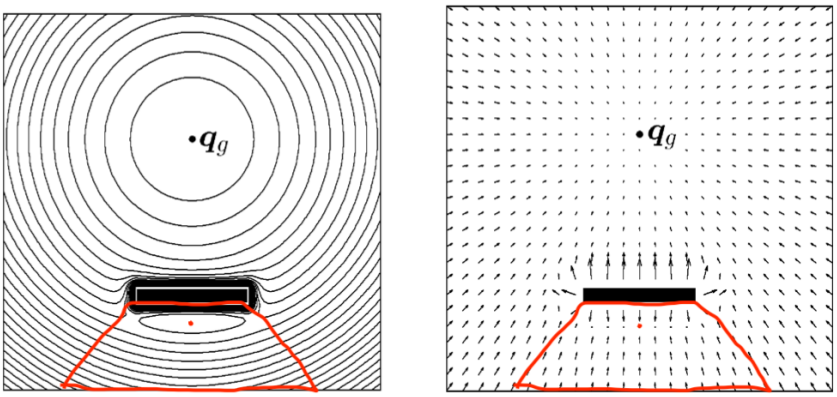

Figure 3: A 2D representation of the total potential and force fields. We can see that the point marked by a red dot becomes a local minimum for both fields. An approximation of the basin of attraction of such minimum is also displayed

4 Code Walkthrough

This section's goal is to provide an accurate description of the enclosed software package available on Github. If you are interested in running the code on a Gazebo simulation, just follow the instructions at `https://github.com/trebeggiani/rl-project`.

For a quick reference to the underlying code structure and data flow, see Fig. 4, where Ros nodes and topics are displayed using the `rqt_graph` app. The `main` Ros node encloses both the control and navigation modules. This choice was taken to reduce Ros communications to a minimum where possible, using multithreading in a single node in place of adopting multiple nodes. The edges connecting the nodes represent Ros topics. The arrow indicates each topic's publisher and subscriber. The pipeline will be presented following its hierarchical structure: starting from the control block, which takes in as inputs only internal (e.g. proprioceptive) measurements and local position goals, and then delving into the higher-level navigation block. Before that, however, a brief introduction on the simulation environment and the robot model will be given.

All relevant source files can be found in the `/popt/src/client/` folder. `Controller.cpp` and `Planner.cpp` deal with the low-level control and the navigation, respectively. Both of them are used by the `main` Ros node in `Main.cpp`.

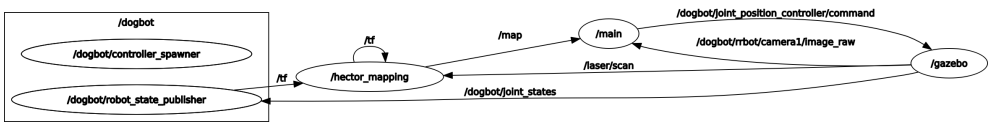

Figure 4: Flowchart of the Ros pipeline drawn by the `rqt_graph` app.

4.1 Simulation environment

The simulation environment that was chosen to test the robot's control and navigation stack is Gazebo. Gazebo is the go-to simulator in the robotics community because of close close integration with Ros and Ros2. In Fact, both the simulator and the system have been developed by Willow Garage (now Open Robotics). Gazebo needs

an environment representation, which can be provided through a `.world` file, and a robot model (one or more than one) as a `.urdf` or `.xacro` file.

Let us focus on the robot model. The `.urdf` model of the React Robotics Dogbot is publicly available [3], along with a set of launch files that can be used to spawn the robot in the desired Gazebo world.

The structure of a `.urdf` or `.xacro` robot model is quite simple. Starting from a base link, joints are added, specifying, for each of them, parent and child links. meshes are also linked for visualization and collision purposes. Fig. 5 displays the `.urdf` model of the robot. Here's an example of a revolute joint attached to the base link (body) of the robot:

```
<!-- roll joint connects the hip to the body -->
  <joint name="back_left_roll_joint" type="revolute">
    <axis rpy="0 0 0" xyz="0 -1 0"/>
    <parent link="body"/>
    <child link="back_left_hip"/>
    <origin rpy="0 0 0" xyz="-0.088 -0.2875 0"/>
    <limit effort="60" lower="-1.7453" upper="1.7453" velocity="6.0"/>
  </joint>
```

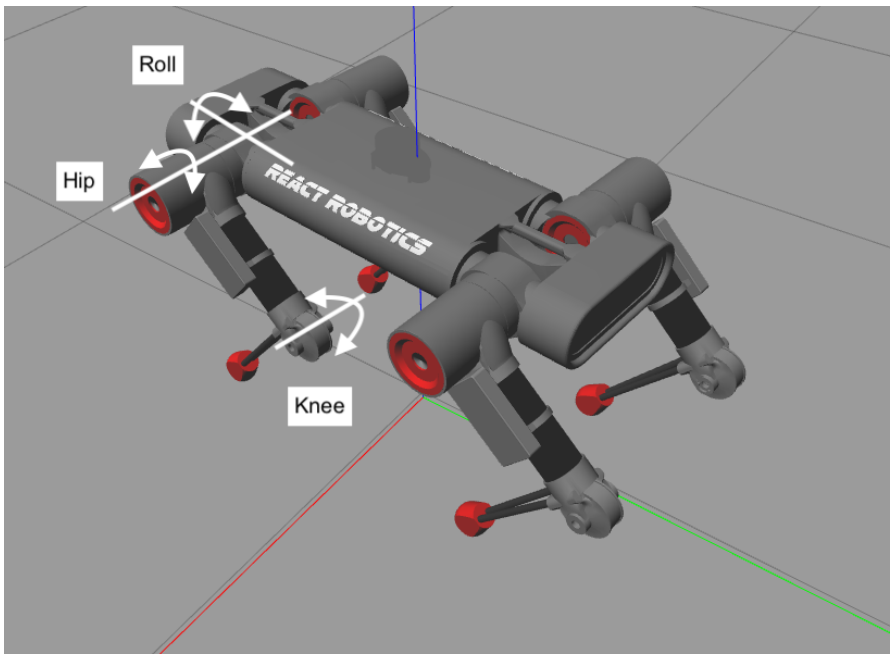

Figure 5: The robot's `.urdf` model, visualized in Gazebo.

To endow the robot with autonomous capabilities, some modifications must be made to the original robot model provided by React Robotics. Namely, a Lidar sensor and a monocular camera are needed. Both can be attached to the robot model simply by adding extra joints and links, and linking a plugin that simulates the sensor and publishes data in Ros. For the sake of simplicity, this procedure is shown here only for the camera sensor.

```
  <link name="RealSense">
    <visual>
      <origin rpy="-1.5707963267948966 0 1.5707963267948966"
      ↪  xyz="-0.05 0 0"/>
      <geometry>
```

[3] GitHub repository of the React Robotics Dogbot quadruped robot `https://github.com/ReactRobotics/DogBotV4`

```
      <mesh filename="package://dogbot_description/meshes/Realsense
      ↪  T265C.stl" scale="0.0015 0.0015 0.0015"/>
    </geometry>
    <material name="black"/>
  </visual>
  <inertial>
    <mass value="0.001"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <inertia ixx="1.6666666666666666e-10" ixy="0.0" ixz="0.0"
    ↪  iyy="1.6666666666666666e-10" iyz="0.0"
    ↪  izz="1.6666666666666666e-10"/>
  </inertial>
</link>
<joint name="RealSense_joint" type="fixed">
  <parent link="body"/>
  <child link="RealSense"/>
  <origin rpy="0 0 -1.5707963267948966" xyz="0 -0.4825 0"/>
</joint>

  <gazebo reference="RealSense">
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>
    <plugin name="camera_controller"
    ↪  filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>rrbot/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera_link</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

As can be seen, the physical link is linked to the mesh of a Realsense depth camera, but the plugin that is used to simulate the camera is that of a simple RGB camera,

as the task only requires the robot to detect and scan QR codes. The plugin allows the user to assign, among other parameters, the image size, frame rate and distortion coefficients.

The launch file in `/dogbot/dogbot_gazebo/launch/dogbot_sim.launch` loads the chosen world as a parameter to the gazebo environment spawner `.launch` file and uses the `dog.launch` file to launch the `urdf_spawner` node in the `gazebo_ros` package.

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find dogbot_gazebo)/worlds/$(arg
    ↪  world)"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="verbose" value="false" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>

  </include>

  <group ns="$(arg namespace)">

    <include file="$(find dogbot_gazebo)/launch/dog.launch">
      <arg name="postgres_log" value="$(arg postgres_log)" />
      <arg name="model" value="$(arg model)" />
      <arg name="paused" value="$(arg paused)"/>
    </include>

  </group>
```

The full simulation setup can be visualized in Figures 6 and 7.

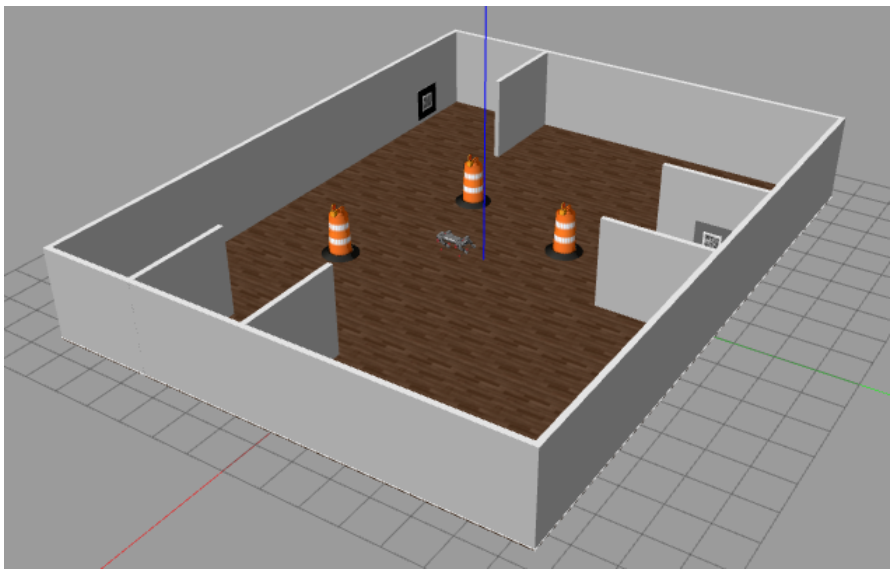

Figure 6: Gazebo GUI view of the simulation scenario.

4.2 Gait Planner

To implement the motion planner in ROS, the Towr library was used [12]. Towr only needs the quadruped model (in our case, the DogBot model), the desired trajectory, and the desired gait (in our case, the *trot* gait will be used). A useful interface to compute the reference trajectories (for both CoM and legs) using Towr was provided by the course instructors with the `get_trajectory` function, whose implementation

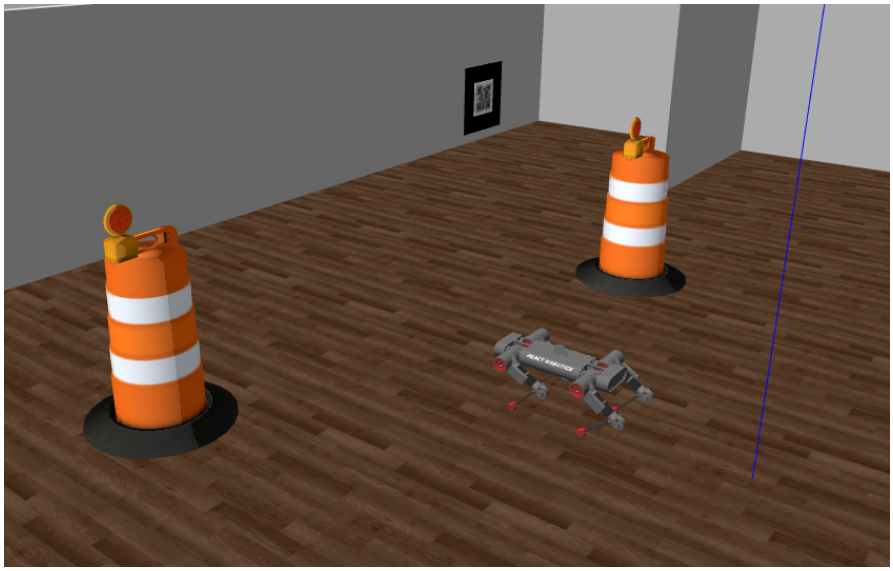

Figure 7: Gazebo GUI view, detail.

can be found in the module `topt`. This function is simply a Towr wrapper that hides the settings that are needed to set the underlying optimization problem solved by Towr.

```
void get_trajectory( Eigen::VectorXd com_p, Eigen::VectorXd com_dp,
↪  Eigen::VectorXd dcom_p, Eigen::Vector3d eeBLpos, Eigen::Vector3d
↪  eeBRpos,
   Eigen::Vector3d eeFLpos, Eigen::Vector3d eeFRpos, int gait_flag,
    ↪  float duration, towr::SplineHolder & solution,
    ↪  towr::NlpFormulation & formulation_ )
```

As can be seen, `get_trajectory()` takes in as input parameters the position and velocity of the robot's CoM, the desired CoM position, the current feet positions, the tipe of gait that we wish the robot to adopt, and the total desired duration of the trajectory. The remaining parameters are output parameters, which respectively contain the points of the splines at each time step and the settings of the optimization solved by Towr. It should be noted that this function uses the pass by value paradigm intentionally, as a safeguard, since it is imperative that parameters do not change during the optimization process, which might take some time.

Once the optimal reference trajectory is obtained via the output parameter `solution`, the references for the CoM and legs can be extracted for each time step and fed to the controller.

4.3 Low-level Control

This module can be found at `/popt/src/client/Controller.cpp` and it deals with computing the actual torque references for the quadruped robot's motors. It is comprised of the low-level planner—that is Towr, in our case—which computes the reference trajectories for both the CoM of the robot and its swing legs, and a controller, that is tasked with following these references as best as possible, optimizing a cost function. The `Controller` class is used by the `main` Ros node, as can be seen in `/popt/src/client/Main.cpp`.

To comply with this task, the module needs to receive measurements from a wide array of sensors. Namely, all joint states (position and velocity) are needed, along with the homogenous transform matrix of the robot base frame w.r.t. the world frame. We need this last one in order to compute the reference wrench that must be applied to the CoM of the robot.

These measurements are collected by the control module via a series of Ros subscribers that listen to simulated sensor data in Gazebo, as such:

```
_joint_state_sub = _nh.subscribe("/dogbot/joint_states", 0,
↪  &Controller::jointStateCallback, this);

_model_state_sub = _nh.subscribe("/gazebo/model_states", 0,
↪  &Controller::modelStateCallback, this);
```

Namely, `_joint_state_sub` listens to joint positions and velocities and processes them with the callback function `Controller::jointStateCallback`. This function simply assigns the joint state values to local member variables in the `Controller` class. This straightforward process is repeated for `_model_state_sub`, which in turn deals with remainder of the robot's state. That is, the base link's position and orientation (along with their derivatives) with respect to the world frame.

The core of the control algorithm can be found in `Controller::ctrl_loop`. This function uses iDyntree to compute the necessary conversions between base and CoM frames. It does so by calling member function `Controller::update`. This function sets a `iDynTree::KinDynComputations` object with the current state measurements and computes the jacobians and transformations that are needed to solve the quadratic problem in (5). `Controller::ctrl_loop` then calls `get_trajectory` to compute the reference trajectory towards the next waypoint, which is a member variable of the `Controller` class set by The navigation module. This step is only executed at start time or whenever the robot reaches a new waypoint. Then the current references are extracted from the sequence computed by Towr, and used to compute the wrench reference as in (9). At this point, the quadratic problem is ready to be set and solved using Alglib [1], an open-source optimizer, as is done in `Controller::qpproblem_stance` and `Controller::qpproblem_swing`. These two functions are used when the feet references are all in the stance phase, or otherwise, respectively. `Controller::qpproblem_swing` checks which feet references are in the swing phase and computes the QP constraints accordingly.

At the end of each iteration, the controller's core loop publishes the torque references that have just been computed, using this Ros publisher:

```
_joint_pub = _nh.advertise<std_msgs::Float64MultiArray>(
↪  "/dogbot/joint_position_controller/command", 1);
```

This topic is in turn subscribed by the Ros motor controllers.

4.4 Navigation

The navigation part of the pipeline can be found in `/popt/src/client/Planner.cpp` and is perhaps the module that requires the most accurate description, as it does not only deal with a simple planning task, but is in charge of using the planner in order to find an user-specified QR code. The `Planner` class is also used by the `main` ros node in `/popt/src/client/Main.cpp`, just like the `Controller`.

Just as a reminder, let us describe the task that should be completed by the robot. The Dogbot should navigate autonomously towards a set of goals assigned by the user and look for a particular QR code, also specified by the user. A location corresponds to the 2D coordinates at the center of a room. The robot must also be able to avoid collisions with obstacles that can be found along the way.

We will start from the higher-level navigation, and then describe the artificial potential fields planner in detail. Like the controller, the navigation module subscribes to different topics. Namely, it needs the current estimate of the base link position and a map of the environment to plan a trajectory. It also needs the images from a camera mounted in the front of the robot, to be able to detect QR codes. The Ros subscribers that handle these messages are the following:

```
map_sub = _nh.subscribe("/map", 1, &Planner::mapCallback, this);

img_sub = _nh.subscribe("/dogbot/rrbot/camera1/image_raw", 1,
↪  &Planner::imageCallback, this);
```

As for the base position, the subscriber is already implemented inside the `Controller` class. Thus, we do not need to create a new subsciber, but we can pass the pointer to the `Controller` object's base position member variable as such:

```
Controller dc(nh);
Planner pl(nh);

// Set base pos reference from the controller
pl.setBasePos(dc.getBasePosPtr());
```

The `/map` topic is published by a separate SLAM system, Hector SLAM [8]. Hector SLAM is a simple lidar-based 2D SLAM system with a long history in the Ros community. Unlike its main competitor system, Gmapping [4], Hector SLAM does not need odometry measurements to provide a state estimate and, most importantly, can be used on robotic platforms that exhibit relevant oscillating behavior, as it also provides (optionally) a convenient IMU-based stabilization module. In Figures 9 and 8 we can visualize the two outputs of the perception module: the occupancy grid map and the estimated 2D pose. The ground truth 2D pose is also present, as a reference.

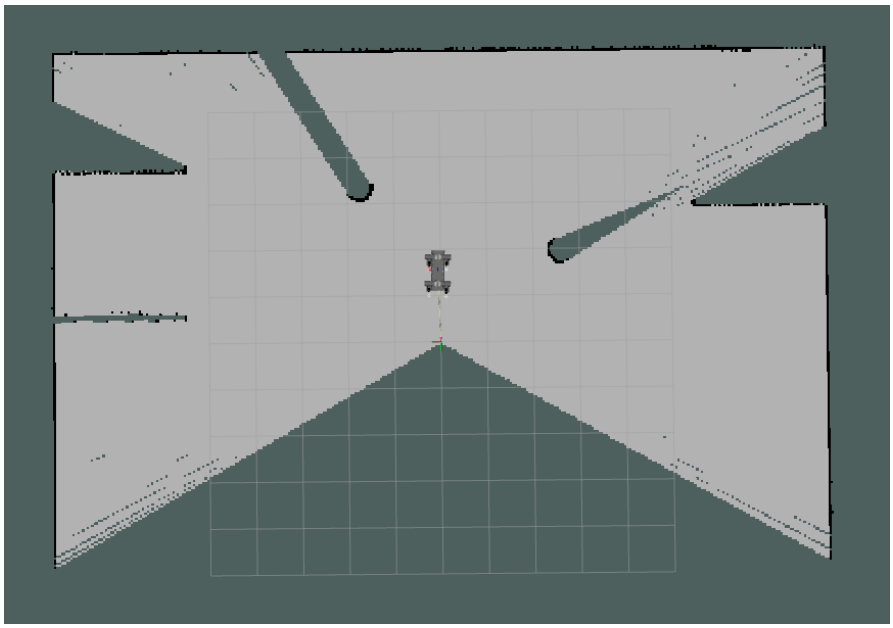

Figure 8: Rviz view of the robot model, along with the visualization of the map reconstruction, and the estimated and ground truth body poses.

While the authors of Hector SLAM also provide a way to extend the 2D SLAM system with a full 6DoF pose, integrating IMU measurements in a separate EKF-based estimation, following an approach inspired by [7], this feature is not used here. There are other effective ways (more effective that general-purpose 3D SLAM) to obtain an accurate estimate of the body pose in quadruped robots [**fallon3**, 13, 9], which fuse leg odometry with lidar, inertial, stereo, and depth maps, but state estimation is not the focus of this project. Therefore, ground truth data is used in place of real full pose estimates. We can better understand how Hector SLAM can be used by looking at selected lines of code from the launch file in `/popt/launch/main.launch`:

```
<node pkg="hector_mapping" type="hector_mapping" name="hector_mapping"
↪  output="screen">
    <param name="scan_topic" value="laser/scan" />
    <param name="output_timing" value="false"/>
    <param name="use_tf_scan_transformation" value="true"/>
    <param name="use_tf_pose_start_estimate" value="false"/>
```

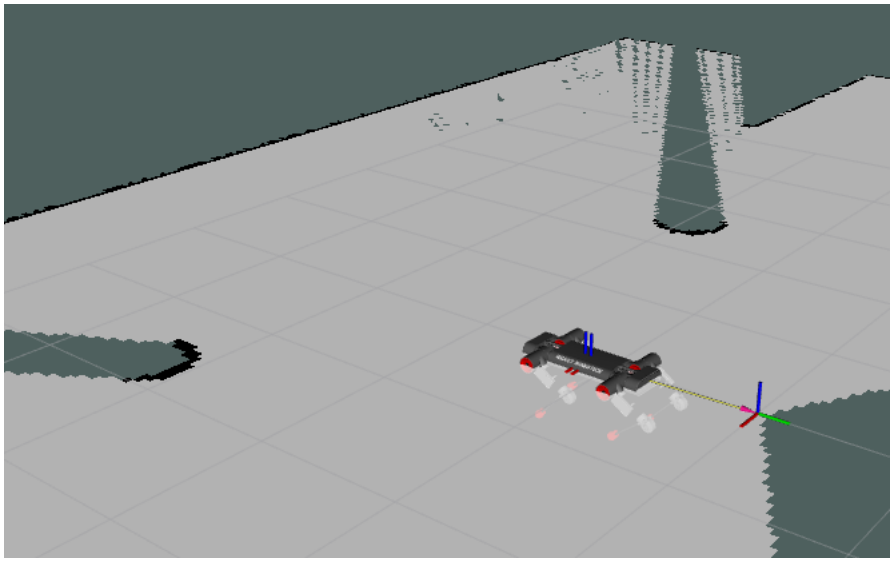

Figure 9: Rviz view, detail.

```
<param name="map_pub_period" value="1.0"/>

<param name="laser_z_min_value" value = "-0.3"/>

<param name="update_factor_free" value="0.3"/>

<param name="map_resolution" value="0.05"/>
<param name="map_size" value="1024"/>
<param name="map_start_x" value="0.5"/>
<param name="map_start_y" value="0.5"/>
<param name="map_multi_res_levels" value="1"/>

<param name="pub_map_odom_transform" value="true"/>
<param name="map_frame" value="map" />
<param name="base_frame" value="base_link" />
<param name="odom_frame" value="base_link" />
</node>
```

By looking at the first parameter in the list, we can see that the node subscribes to the `laser/scan` topic. Scans are transformed by the `hector_mapping` node into the `base_link` frame by looking for the relative static Ros `tf`. Scans are then matched to produce a 2D occupancy grid and estimate the robot pose. This estimate is produced by the node in the form of a Ros `tf` between the frames `base_link` and `map` (this last one is just an arbitrarily assigned name to use as a reference frame). The SLAM system is in fact assigning a parent-child relation between such frames.

Now, let us see how these data are used by the higher-level part of the control infrastructure. The core of the navigation module is found in the following member function in the `Planner` class:

```
void Planner::master()
{
  bool found = false; // true if we found the target QR code
    while (ros::ok())
  {
    // pop next goal from goal vector
    current_goal = goals.back();
    goals.pop_back();
```

```
    // go to next goal
    artpot();
    // look around, try to find the qr code
    found = findQR();

    // now we're ready to head to next room (or home)

    // if there are no goals left, or target found --> go home
    if(goals.empty() || found) {
      current_goal = home;
      artpot(); // go home
      break; // exit while loop -> end navigation
    }
  }

}
```

What the `master` function does at each iteration is it pops the next goal from an array of 2D coordinates and navigates towards it using the artificial potential fields method. Once the robot reaches the goal, it spins around the center of the room, looking for a QR code. If a QR code is found, and if it is the right one, the robot goes home. In all other cases, the robot goes to the next location. If the array of locations is empty, the robot goes home. We can see that the goals are stored in a `std::vector<Eigen::Vector2d>` container. The goal coordinates vector and the home coordinates are set in the `main` function at `/popt/src/client/Main.cpp`, using set functions from the `Planner` class.

5 Experimental Results

In the following, we review the results of the Gazebo simulation where the software was tested. Only the results relative to the low-level controller part are reviewed here. To verify the functionality of the navigation module, please watch the enclosed demo video in the `media` folder. In the same folder, you will also find a short video of a simulation focusing on the robustness of the QP controller, where the robot's body link is subject to impulsive forces of increasing magnitude.

All of the results that are presented here are obtained by recording variables during a straightforward straight walk along the x axis of the global reference frame. The chosen Towr gait is the "trot" gait. The step length, which is a parameter of the artificial potential fields algorithm, was fixed at a maximum of $5cm$. Only a few seconds of this walk are extracted to display variables clearly. The reference variables are the ones computed by the Towr motion planner. The measurements, instead, are extracted from the internal state representation that the robot controller constructs from the sensor suite, using the iDyntree library.

In Figure 10 we can observe the evolution of the CoM position, along with the reference computed by Towr. The oscillation of the CoM on the z-axis is due to the underlying optimization process inside Towr: to generate smoother trajectories, the planner also exploits the body link elevation. The deviation from the nominal height is a parameter that can be set in the Towr configuration file. The controller appears to have sufficient tracking properties, as the torques that are fed to the motor controllers reduce the error norm to peaks that stay below $5mm$. The CoM position error and its moving average over $2s$ are displayed in Figs. 13 and 14.

Figure 11 displays the back-right foot position measurement and reference against time. The swing phase is clearly visible in the x-axis and z-axis subplots. The foot schedule has a "duty cycle" of exactly 1/4, as it is stationary for 75% of the time. The foot error is also kept very low, as it reaches $1cm$ in very few occasions. Such harmless occasions coincide with sporadic delays in the controller torque computation. The back-right foot position error and its moving average over $2s$ are shown in Figs. 13 and 14.

Fig. 11 shows the gait of the robot at the feet level. That is, the X-Z foot trajectory. The figures show how the controller's swing leg gains manage to allow for a smooth reference following in the sagittal plane. A measure of the elapsed time at each point is not given, but the purpose of this plot is to provide a clear planar visualization of the gait. For the sake of clarity, the plot only displays the back-right foot position.

In Figure **??** we can see how the control torques evolve. The torques that are reported are those from the joints of the back-right leg. Figure 5 illustrates where the roll, hip, and knee joints are situated on the robot model. As can be seen, the torque norm remains under $10N$ for the roll and hip joints, while it reaches above $20N$ for the knee joint. Both values are well below the saturation limit of the robot's actuators, which is set at $60N$ in the robot `.xacro` model.

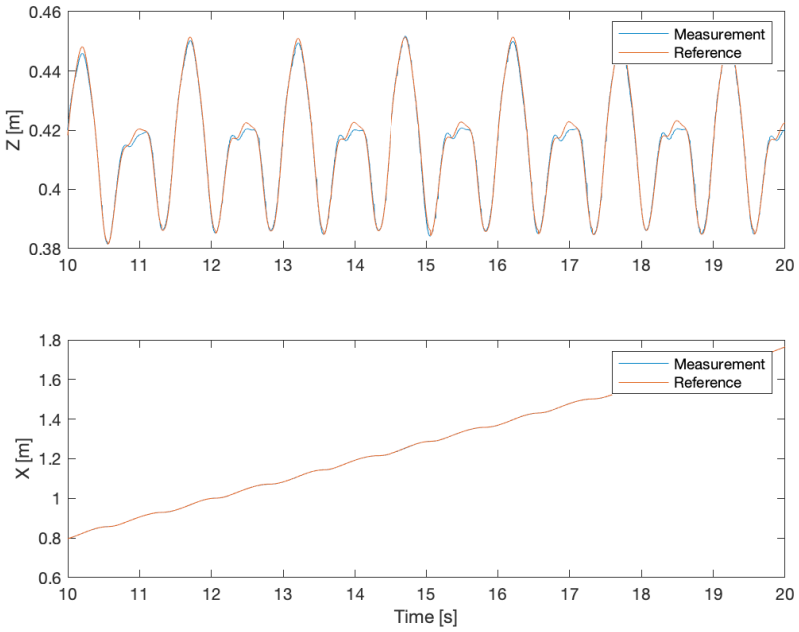

Figure 10: CoM position against time.

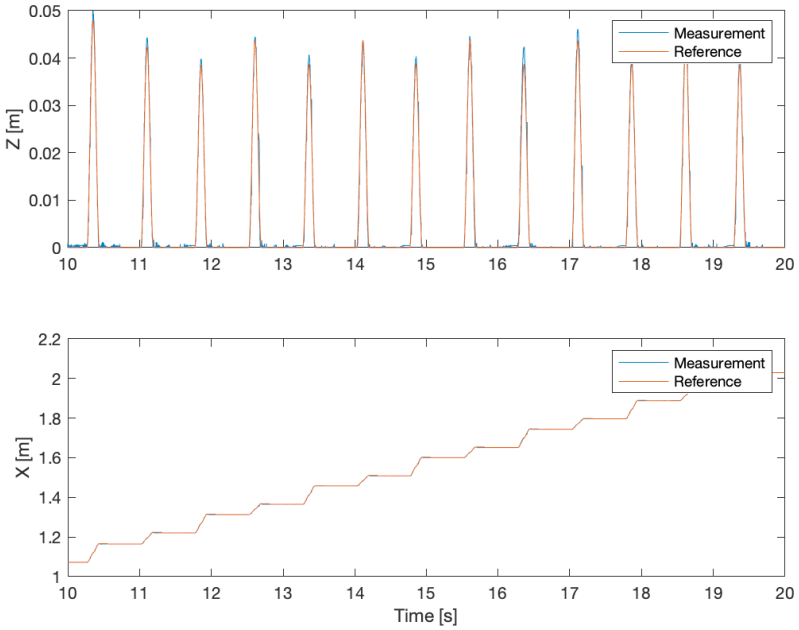

Figure 11: Back right foot position against time.

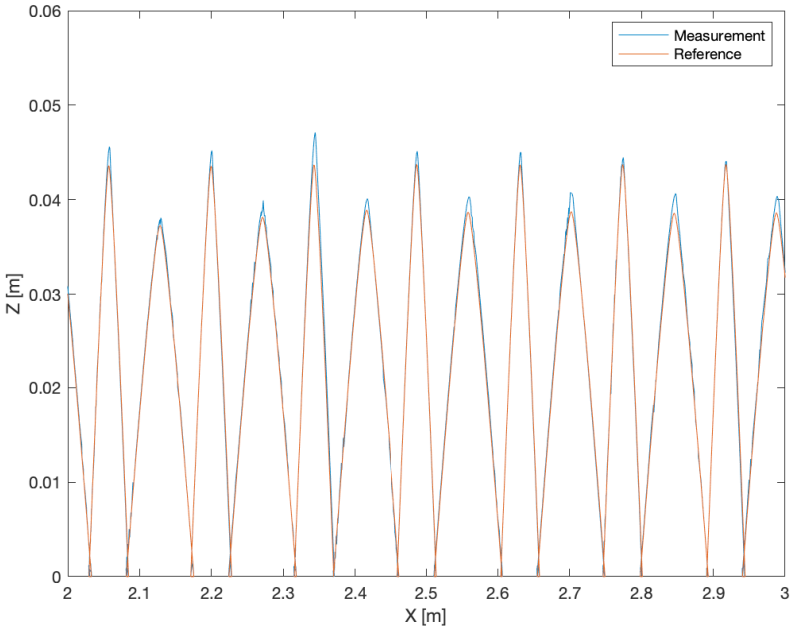

Figure 12: Back right foot position in the X-Z plane.

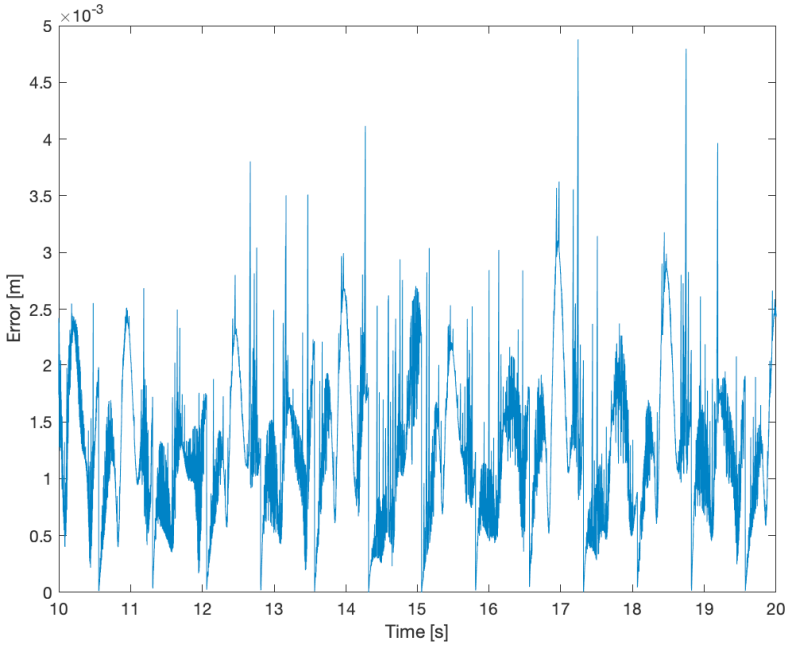

Figure 13: CoM position error against time.

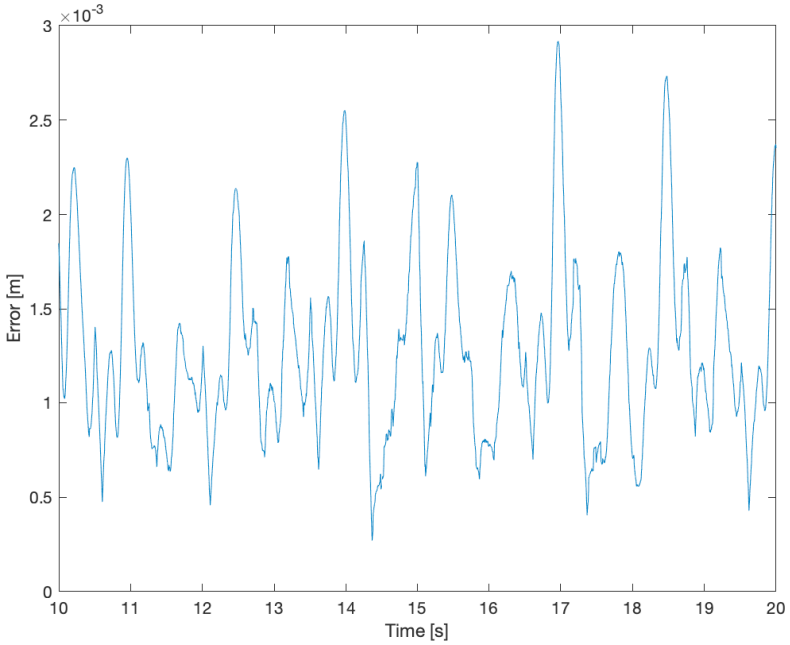

Figure 14: CoM position error against time (MA-filtered over $2s$).

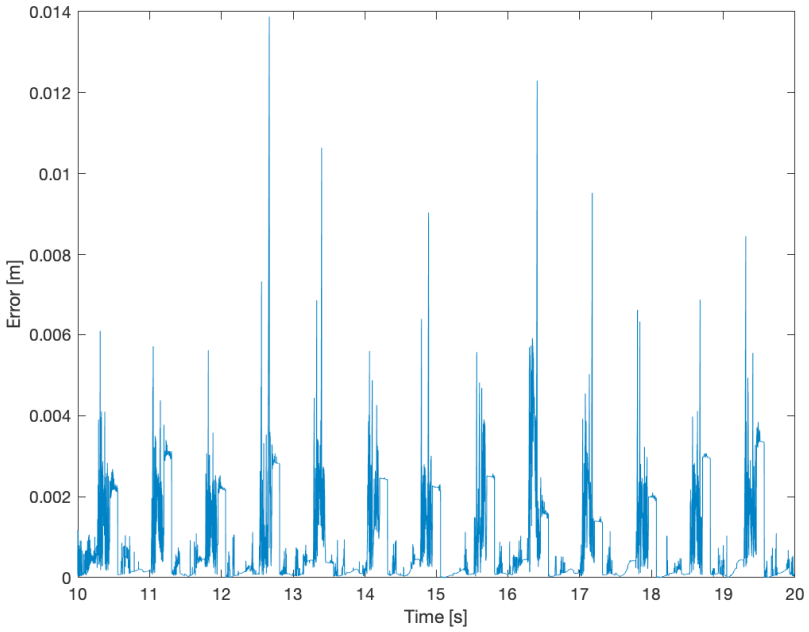

Figure 15: Back right foot position error against time.

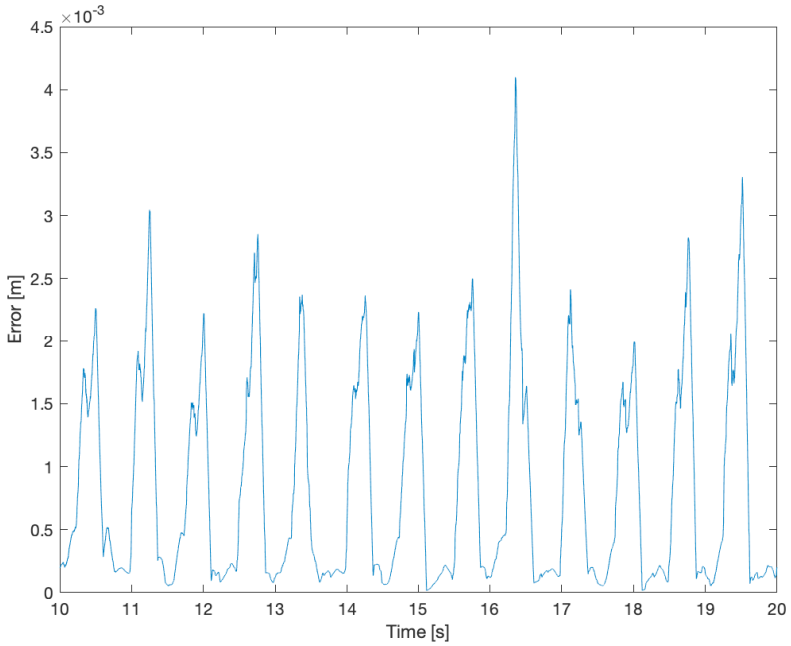

Figure 16: Back right foot position error against time (MA-filtered over $2s$).

6 Concluding Remarks

This report has given a concise overview on the process of developing a controller for a complex robotic system, and testing it on a simulation platform using Ros and Gazebo. Although much of the effort that usually goes into building such system is usually devoted to the practical implementation details, it is imperative to never disregard the theoretical background behind the algorithms that we use. This is why a consistent part of this work focuses just on that, as an exercise. Perhaps the biggest lesson learned here is that strictly non-learning methods still often manage to perform adequately, even when subject to external disturbances.

Future development. A great improvement, that surely would help bring this project one step closer to a real-world-capable robot control infrastructure, would be to endow the robot with true state estimation capabilities, following one of the state-of-the art methods discussed above, in section 3.

References

[1] ALGLIB - numerical analysis library. *About ALGLIB*. URL: `https://www.alglib.net`.

[2] C Dario Bellicoso et al. "Dynamic locomotion and whole-body control for quadrupedal robots". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 3359–3365.

[3] Shamel Fahmi et al. "Passive whole-body control for quadruped robots: Experimental validation over challenging terrain". In: *IEEE Robotics and Automation Letters* 4.3 (2019), pp. 2553–2560.

[4] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved techniques for grid mapping with rao-blackwellized particle filters". In: *IEEE transactions on Robotics* 23.1 (2007), pp. 34–46.

[5] Jemin Hwangbo et al. "Learning agile and dynamic motor skills for legged robots". In: *Science Robotics* 4.26 (2019), eaau5872.

[6] Oussama Khatib. "Real-time obstacle avoidance for manipulators and mobile robots". In: *Autonomous robot vehicles*. Springer, 1986, pp. 396–404.

[7] Alexander Kleiner and Christian Dornhege. "Real-time localization and elevation mapping within urban search and rescue scenarios". In: *Journal of Field Robotics* 24.8-9 (2007), pp. 723–745.

[8] Stefan Kohlbrecher et al. "A flexible and scalable SLAM system with full 3D motion estimation". In: *2011 IEEE international symposium on safety, security, and rescue robotics*. IEEE. 2011, pp. 155–160.

[9] Simona Nobili et al. "Heterogeneous Sensor Fusion for Accurate State Estimation of Dynamic Legged Robots." In: *Robotics: Science and Systems*. 2017.

[10] Fabio Ruggiero Viviana Morlando Ainoor Teimoorzadeh. "Whole-body control with disturbance rejection through a momentum-based observer for quadruped robots". In: *Mechanism and Machine Theory* (2021).

[11] Miomir Vukobratovic and Davor Juricic. "Contribution to the synthesis of biped gait". In: *IEEE Transactions on Biomedical Engineering* 1 (1969), pp. 1–6.

[12] Alexander W Winkler et al. "Gait and Trajectory Optimization for Legged Systems through Phase-based End-Effector Parameterization". In: *IEEE Robotics and Automation Letters (RA-L)* 3 (July 2018), pp. 1560–1567. DOI: `10.1109/LRA.2018.2798285`.

[13] David Wisth, Marco Camurri, and Maurice Fallon. "Robust legged robot state estimation using factor graph optimization". In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4507–4514.