Field and Service Robotics - Technical Report

Model-based and Data-driven Control
of a Quadruped Robot

Group:
Emanuele Cuzzocrea
Vito Daniele Perfetta

Chapter 1

Introduction

This technical report presents an analysis regarding the results of model-based and data-driven control approaches on legged robots. In particular, an MPC approach is analyzed in the former case, while a Deep Reinforcement Learning technique based on the Reward Machines is developed in the latter. The robot behavior is tested using different gaits with both the approaches.

Moreover, exploiting the model-based control, an RRT planning algorithm is implemented to see how the robot moves in an environment with obstacles.

All the results are obtained in ROS using Gazebo and Rviz environments for the model-based approach and the RaiSim platform for the data-driven one. In the first case, the simulation is performed with the Unitree A1 Robot Dog, while in the second with the Unitree Aliengo Robot Dog.

The produced code and video regarding the results discussed in this report are available in the related public repository on github `https://github.com/danperf/FieldAndServiceRobotics.git`.

(a) Unitree A1 Robot Dog (b) Unitree Aliengo Robot Dog

Figure 1.1

Chapter 2

Model-based Control

The model-based approach is a control strategy that explicitly uses the mathematical model of the considered system to predict its behavior and make informed control decisions. As anticipated, the analyzed model-based approach is the whole-body MPC. The key advantage of this control technique is the capability of managing a multi-variable system while taking into account several constraints on the environment and on the robot itself.

It has been chosen to start from the implementation already provided in the attached public github repository[1], which will be properly modified as discussed in Section 2.2 "Gait analysis".

2.1 Model Predictive Control

From [3] and [4], the overall control architecture of the considered system is depicted in the block diagram in Fig 2.1.

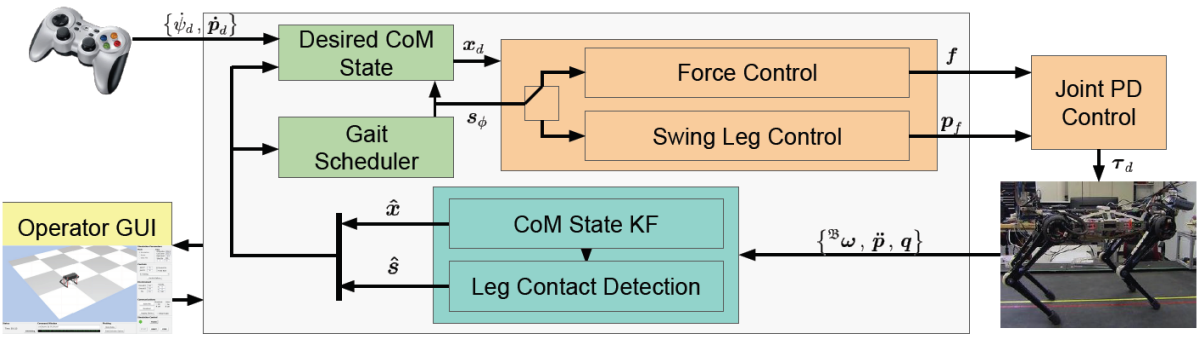


Figure 2.1: System Architecture Block Diagram

The operator provides high-level commands by giving a desired translational velocity $\dot{p}_d$ and turning rate $\dot{\psi}_d$. These commands are used to plan a smooth and controllable CoM reference trajectory that is relayed to the body and leg controllers. The gait is defined by an event-based finite state machine that uses a leg-independent phase variable to schedule nominal contact and swing phase.

The controller proposed in these papers determines desired 3-dimensional ground reaction forces for behaviors with fixed-timing foot placement and liftoff. When a foot is scheduled to be in the air, the robot runs a swing leg controller. Otherwise, a ground force controller runs. Each of the robot's four legs has three torque controlled joints: calf, knee and thigh. By design, the robot has light limbs with low inertia as compared to the overall body. For this reason, the control model can be simplified such to ignore the effects of the legs for planning ground reaction forces from the stance feet.

The swing controller computes and follows a trajectory for the foot in the world coordinate system.

$$\tau_i = J_i^T \left[K_p(p^b_{i,ref} - p^b_i) + K_d(v^b_{i,ref} - v^b_i) \right] + \tau_{i,ff} \tag{2.1}$$

where J_i is the foot Jacobian, K_p and K_d are diagonal positive definite proportional and derivative gain matrices, p^b_i and v^b_i are the position and velocity of the i-th foot in the body frame, $p^b_{i,ref}$ and $v^b_{i,ref}$ are the corresponding references for the position and velocity of the swing leg trajectory and $\tau_{i,ff}$ is a feedforward torque

$$\tau_{i,ff} = J_i^T \Lambda_i (a^b_{i,ref} - \dot{J}_i \dot{q}_i) + C_i \dot{q}_i + G_i \tag{2.2}$$

where Λ_i is the operational space inertia matrix, $a^b_{i,ref}$ is the reference acceleration in the body frame, q_i is the 3-dimensional vector of joint positions, and $C_i\dot{q}_i + G_i$ is the torque due to gravity and Coriolis forces for the leg.

During ground force control, joint torques are computed with

$$\tau_i = J_i^T R_i^T f_i \tag{2.3}$$

where R_i is the rotation matrix which transforms from foot to world coordinates and f_i is the vector of forces calculated from the model predictive controller.

Neglecting the leg dynamics, the simplified robot dynamic model is

$$\frac{d}{dt}\begin{bmatrix}\hat{\Theta}\\ \hat{p}\\ \hat{\omega}\\ \hat{\dot{p}}\end{bmatrix} = \begin{bmatrix}0_3 & 0_3 & R_z(\psi) & 0_3\\ 0_3 & 0_3 & 0_3 & 1_3\\ 0_3 & 0_3 & 0_3 & 0_3\\ 0_3 & 0_3 & 0_3 & 0_3\end{bmatrix}\begin{bmatrix}\hat{\Theta}\\ \hat{p}\\ \hat{\omega}\\ \hat{\dot{p}}\end{bmatrix} + \begin{bmatrix}0_3 & ... & 0_3\\ 0_3 & ... & 0_3\\ \hat{I}^{-1}[r_1]_\times & ... & \hat{I}^{-1}[r_n]_\times\\ 1_3/m & ... & 1_3/m\end{bmatrix}\begin{bmatrix}f_1\\ ...\\ ...\\ f_n\end{bmatrix} + \begin{bmatrix}0\\ 0\\ 0\\ g\end{bmatrix} \tag{2.4}$$

where p is the robot's position, m is the robot's mass, g is the acceleration of gravity, $\hat{I}$ is the robot's inertia tensor, $\Theta = [\phi, \theta, \psi]^T = [pitch, roll, yaw]^T$ expresses the robot's orientation, ω is the robot's angular velocity, R is the rotation matrix which transforms from body to world coordinates, $[x]_\times$ is defined as the skew-symmetric matrix such that $[x]_{\times y} = x \times y$, f_i is the ground reaction force with n representing the robot's number of legs and r_i is the vector from the center of mass (COM) to the point where the force is applied. This result is obtained under the assumption of having small values of roll and pitch. Equation (2.4) can be rewritten in the following state-space form adding the gravity term in the states

$$\dot{x}(t) = A_c(\psi)x(t) + B_c(r_1, ..., r_n, \psi)u(t) \tag{2.5}$$

where $x(t) = \left[\hat{\Theta}, \hat{p}, \hat{\omega}, \hat{\dot{p}}, g\right]^T$, $A_c \in \mathbb{R}^{13\times 13}$, $B_c \in \mathbb{R}^{13\times 3n}$ and the control inputs $u(t)$ are the ground reaction forces. This form depends only on yaw and footstep locations. If these can be computed ahead of time, the dynamics become linear time-varying, which is suitable for convex model predictive control. Since the ground reaction forces are computed instead of joint torques, the predictive controller does not need to be aware of the configuration or kinematics of the leg.

Eventually, desired ground reaction forces are found with a discrete time finite-horizon model predictive controller where the desired values of ψ and r_i from the reference trajectory and foot placement controller are used to compute the discrete version of A_c and B_c (in the equation (2.7) written below, they are simply renamed as A_i and B_i). This approximation holds only if the robot is able to follow the reference trajectory.

$$\min_{x,u} \sum_{i=0}^{k-1} ||x_{i+1} - x_{i+1,ref}||_{Q_i} + ||u_i||_{R_i} \tag{2.6}$$

$$s.t.$$

$$x_{i+1} = A_i x_i + B_i u_i \tag{2.7}$$

$$\underline{c}_i \leq C_i u_i \leq \overline{c}_i \tag{2.8}$$

$$D_i u_i = 0 \tag{2.9}$$

The equality constraint in (2.9) is used to set all forces from feet off the ground to zero, enforcing the desired gait. The inequality constraint in (2.8) is used to set the following 10 inequality constraints for each foot on the ground

$$f_{min} \leq f_z \leq f_{max} \tag{2.10}$$

$$-\mu f_z \leq \pm f_x \leq -\mu f_z \tag{2.11}$$

$$-\mu f_z \leq \pm f_y \leq -\mu f_z \tag{2.12}$$

These constraints limit the minimum and maximum z-force as well as a square pyramid approximation of the friction cone. To lighten the MPC's computational demand, it's possible to remove the system dynamics from the constraints and include them in the cost function instead. Rewriting the dynamics as

$$X = A_{qb} x_0 + B_{qp} U \tag{2.13}$$

where $X \in \mathbb{R}^{13k}$ is the vector of all states during the prediction horizon and $U \in \mathbb{R}^{3nk}$ is the vector of all control inputs during the prediction horizon. The objective function which minimizes the weighted least-squares deviation from the reference trajectory and the weighted force magnitude is

$$J(U) = ||A_{qb} x_0 + B_{qp} U - x_{ref}||_L + ||U||_K \tag{2.14}$$

with $L = diag_k(Q)$ and $K = diag_k(R)$. The problem can be rewritten as

$$\min_U \frac{1}{2} U^T H U + U^T g \tag{2.15}$$

$$s.t.$$

$$\underline{c} \leq CU \leq \overline{c} \tag{2.16}$$

where C is the constraint matrix and

$$H = 2(B_{qp}^T L B_{qp} + K) \tag{2.17}$$

$$g = 2B_{qp}^T L(A_{qp} x_0 - x_{ref}) \tag{2.18}$$

The desired ground reaction forces are then the first $3n$ elements of U.Here, the constraints (2.9) aren't required anymore since the control optimize only forces for stance feet.

In the following application, the reference trajectories only contain non-zero xy-velocity, xy-position, z position, yaw, and yaw rate. All parameters are commanded directly by the robot operator except for yaw and xy-position, which are determined by integrating the appropriate velocities. The other states (roll, pitch, roll rate, pitch rate, and z-velocity) are always set to 0.

2.2 Gait analysis

The considered repository[1] allows to test the discussed controller only with trot, crawl and bound gaits. Assigning only a desired velocity along the x-axis, the corresponding results are shown below. In the following analysis, the front left leg is appointed as leg "1", the front right one as "2", the rear left leg as "3" and the rear right one as "4".

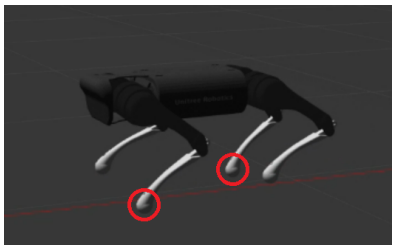


Figure 2.2: Trot gait

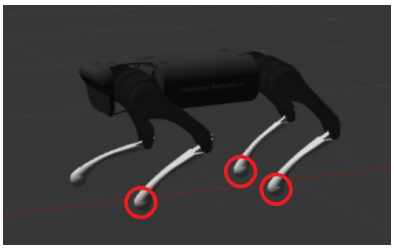


Figure 2.3: Crawl gait

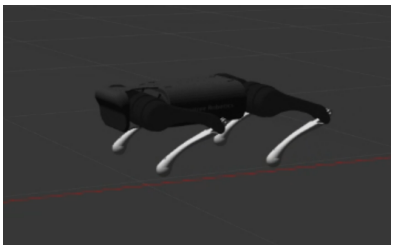
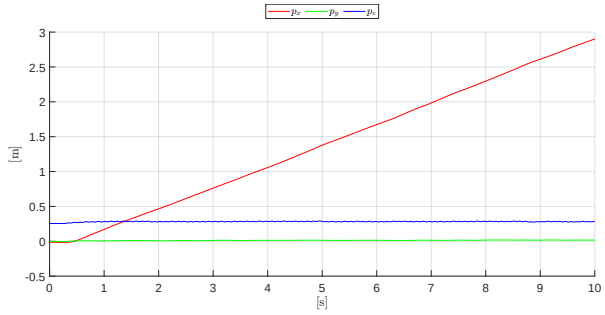
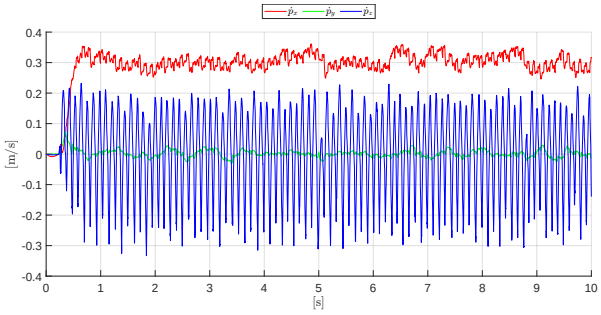


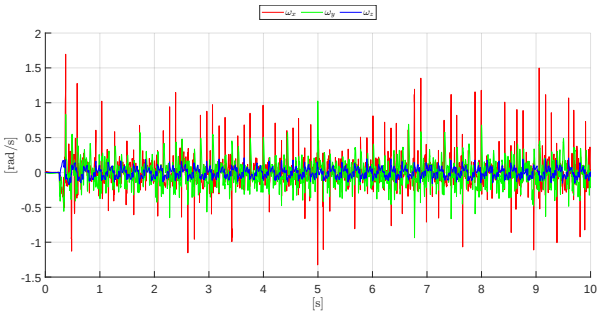
Figure 2.4: Old bound gait



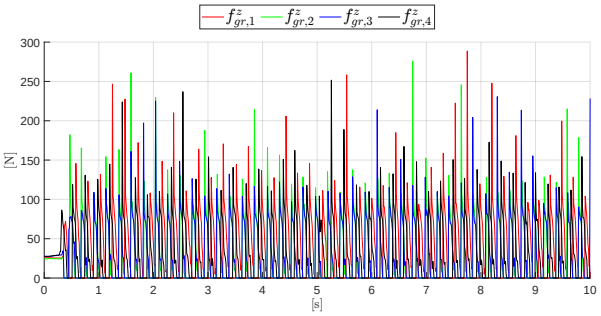
(a) position



(b) velocity

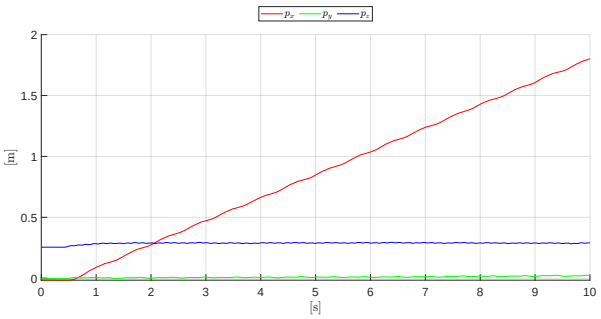


(c) angular velocity

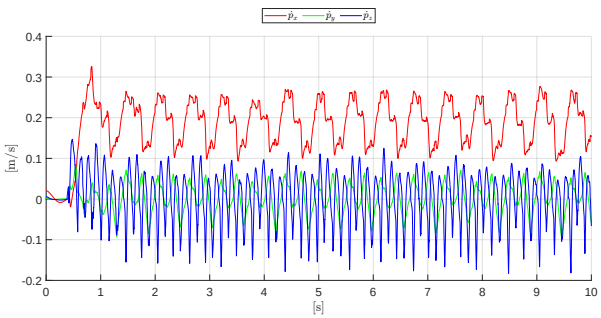


(d) GRF_z

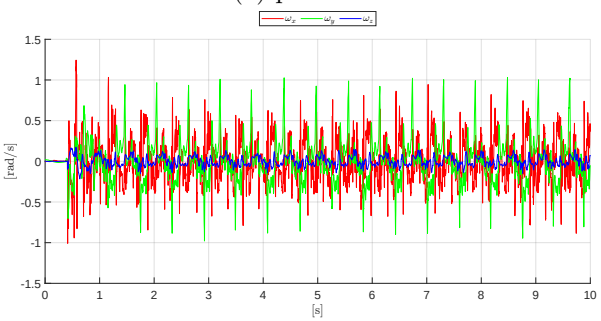
Figure 2.5: trot, $v_{x,d} = 0.3m/s$



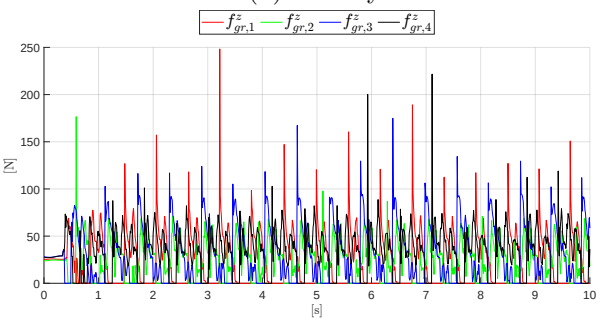
(a) position



(b) velocity

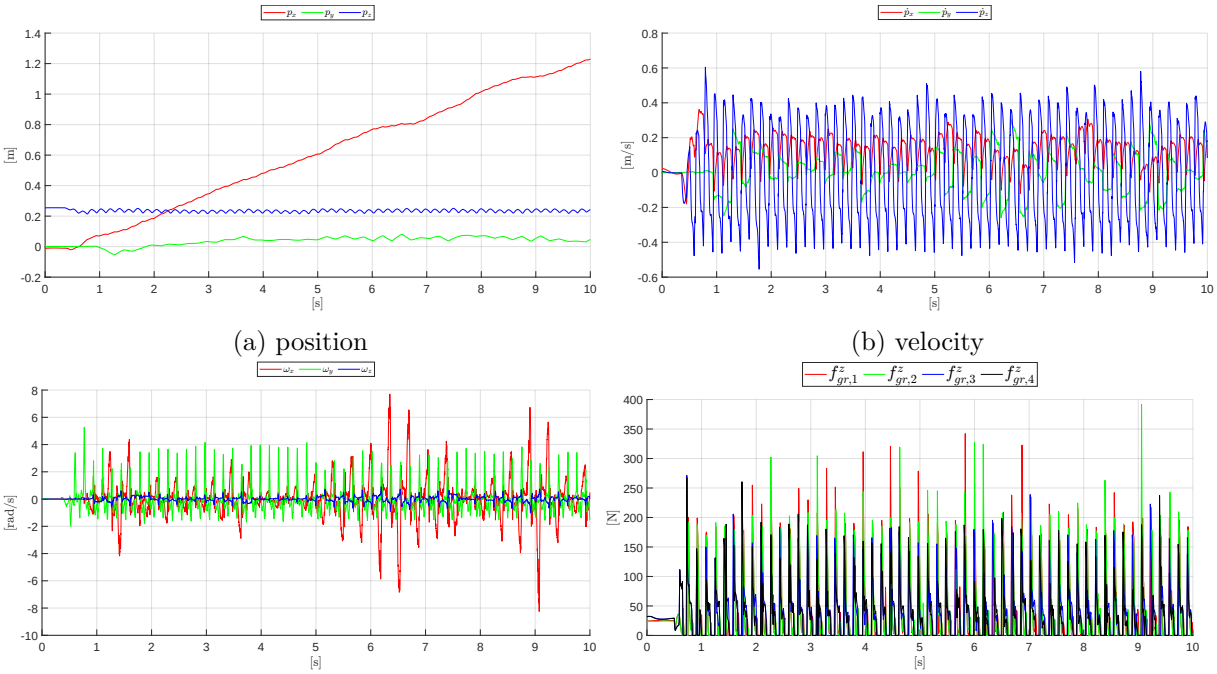


(c) angular velocity



(d) GRF_z

Figure 2.6: crawl, $v_{x,d} = 0.2m/s$



(a) position

(b) velocity

(c) angular velocity

(d) GRF_z

Figure 2.7: old bound, $v_{x,d} = 0.3m/s$

By analyzing the ground reaction forces in the different gaits it's possible to notice which are the feet in contact with the ground in each instant of time. For instance, looking at Fig 2.5.d, the robot walks having alternatively the feet 2-3 and 1-4 in contact with the ground. On the other hand, there are at least three stance feet in each instant of time during the crawl gait, as shown in Fig 2.6.d. In the bound gait, the behavior is different from the traditional case since all the legs are jumping quite at the same time, as shown in Fig 2.7.d, instead of moving in pairs. This creates also an oscillation along the z-axis of the center of mass, as shown in Fig 2.7.a,b.
As it can be noticed, even if in every case the motion is stable, the performance reached by the control are different according to the used gait. For instance, with the current version of the bound gait, the robot isn't able to achieve the reference linear velocity, but there is an offset along the x-axis.

This analysis can be extended also for other new implemented gaits, changing the code in the proper way. In particular, the pace and gallop have been added, while the bound has been modified such to make front and rear legs jump alternatively.
To do this, the following lines have been added in "openloop_gait_generator.yaml" file inside the folder associated to the MPC demo.

```
...
bound:
    stance_duration: [ 0.3, 0.3, 0.3, 0.3]
    duty_factor: [ 0.5, 0.5, 0.5, 0.5]
    init_phase_full_cycle: [ 0., 0., 0.9, 0.9]
    initial_leg_state: [ 0, 0, 1, 1]
    contact_detection_phase_threshold: 0.1
  pace:
    stance_duration: [0.2, 0.2, 0.2, 0.2]
    duty_factor: [0.5, 0.5, 0.5, 0.5]
```

```
    init_phase_full_cycle: [1., 0., 1., 0.]
    initial_leg_state: [1, 0, 1, 0]
    contact_detection_phase_threshold: 0.1
  gallop:
    stance_duration: [0.2, 0.2, 0.2, 0.2]
    duty_factor: [0.5, 0.5, 0.5, 0.5]
    init_phase_full_cycle: [0., 0.4, 0.8, 0.2]
    initial_leg_state: [1, 1, 1, 1]
    contact_detection_phase_threshold: 0.1
...
```

where

- stance_duration specifies the amount of stance time for each leg in a gait cycle.
- duty_factor represents the fraction of stance phase in the gait cycle.
- init_phase_full_cycle is the initial phase of each leg at the beginning of the full cycle.
- initial_leg_state is the initial state of each leg, where 1 indicates "STANCE" and 0 indicates "SWING".
- contact_detection_phase_threshold specifies the contact threshold when the leg state switches from "SWING" to "STANCE".

In particular, the init_phase_full_cycle and the init_leg_state are the key factors in order to select which legs move together (e.g. bound and pace) and/or which is the legs' motion order (e.g. gallop) to create the desired gait. Moreover, the symmetric behavior desired in bound and pace is ensured assigning the duty_factor's value around 0.5. On the other hand, the fast motion characterizing the gallop gait requires smaller values of the stance_duration parameters. These values are then used by the foothold planner and by swing and stance legs controllers in order to implement the desired feet trajectories.

An additional modification has been made to the matrix Q that weights the states in order to achieve a better trajectory tracking along the x-axis and to contain potential drift along the y-axis. Q has been set equal to $[10, 10, 5, 60, 60, 100, 0., 0., 0.5, 50, 100, 1, 0]$ changing the weights related to p_x, $\dot{p}_x$ and $\dot{p}_y$ in such a way to obtain satisfactory results with all the analyzed gaits. The matrix R hasn't been changed, maintaining the previous setting $R = 4 \cdot 10^{-6} I_{3n}$.

It's possible to open the world and start the controller respectively using

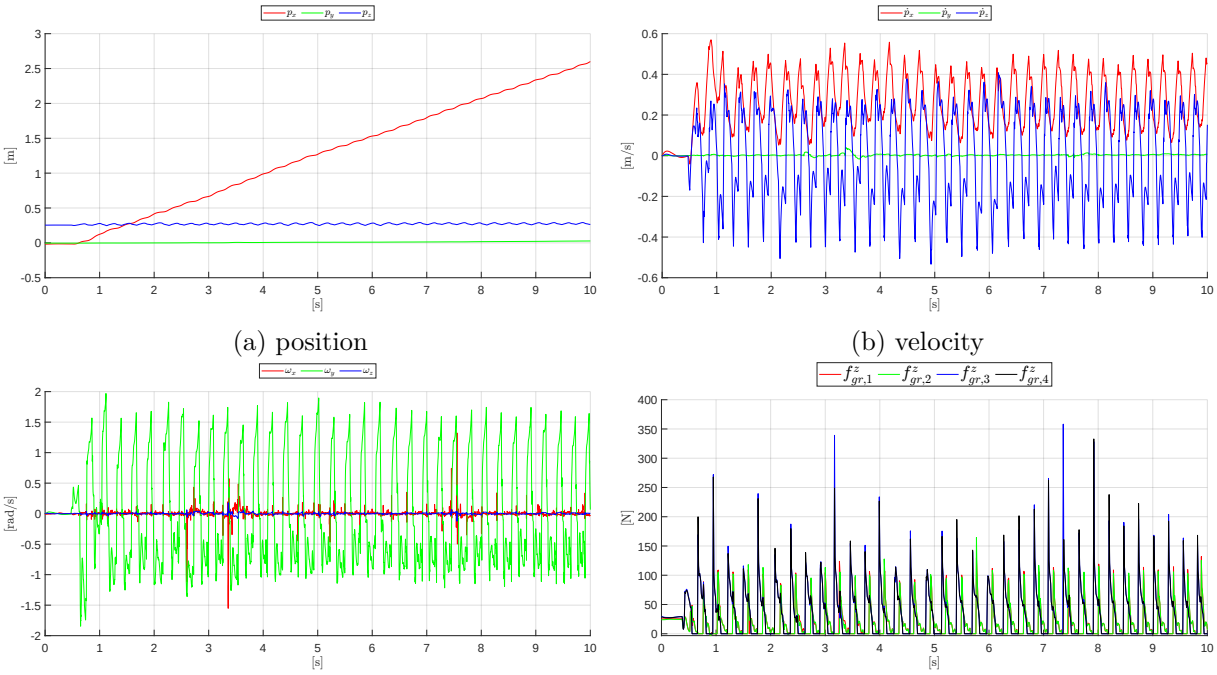
```
$ roslaunch unitree_gazebo normal.launch rname:=a1 use_xacro:=
   true
```

```
$ rosrun demo demo_trot_velocity_mpc
```

The corresponding results have been reported below.

In Fig 2.8, it's possible to notice how the new version of the bound gait is such that foot 1 jumps together with 2, while 3 with 4. Due to these jumps, there are high oscillations of the linear velocity along the x-axis as well as the angular velocity along the y-axis, as shown respectively in Fig 2.8.b and Fig 2.8.c. Moreover, also the center of mass moves up and down as shown in Fig 2.8.a,b. In this case, the robot is able to achieve the desired linear velocity along x. From Fig 2.9.d it's possible to notice that the pairs of alternate stance feet in pacing gait are 1-3 and 2-4. Since the robot moves alternatively the legs on the same side, an high oscillation of the linear velocity occurs along the y-axis, as shown in Fig 2.9.b. The same happens with the angular velocity along the x-axis, Fig 2.9.c. The gallop gait is characterized by one peak

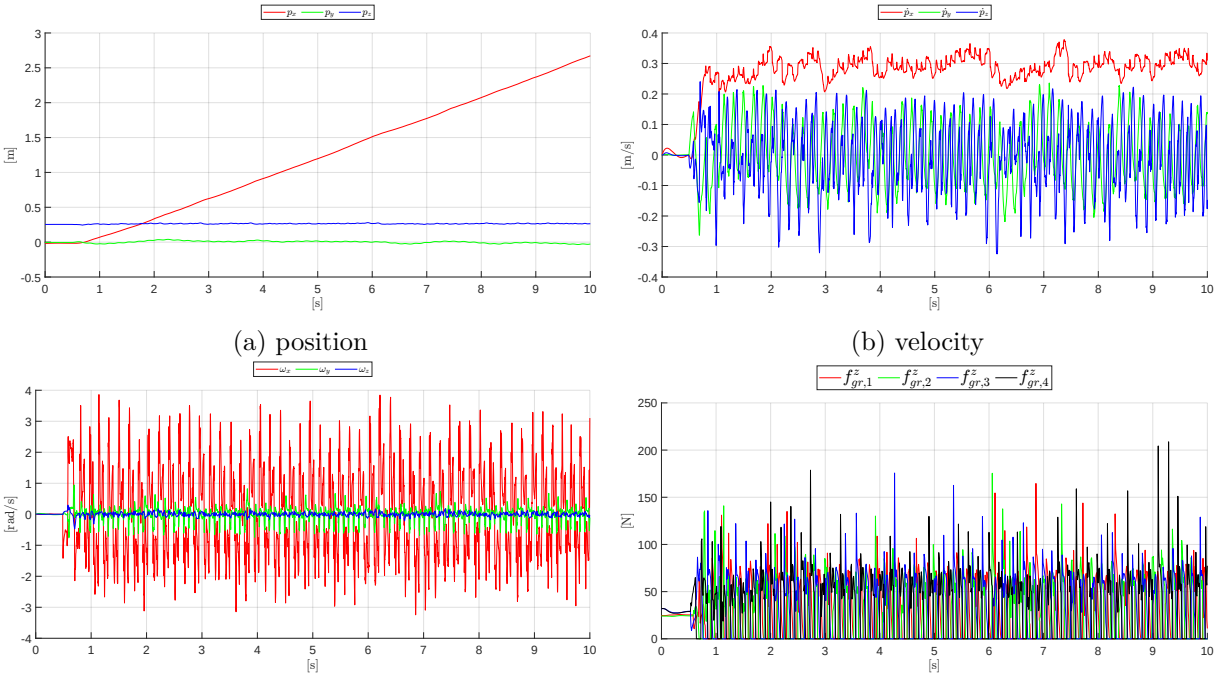
of ground reaction force at time, as shown in Fig 2.10.d. Moreover, the robot exhibits high oscillations of linear and angular velocities respectively along y and x.



(a) position (b) velocity

(c) angular velocity (d) GRF_z

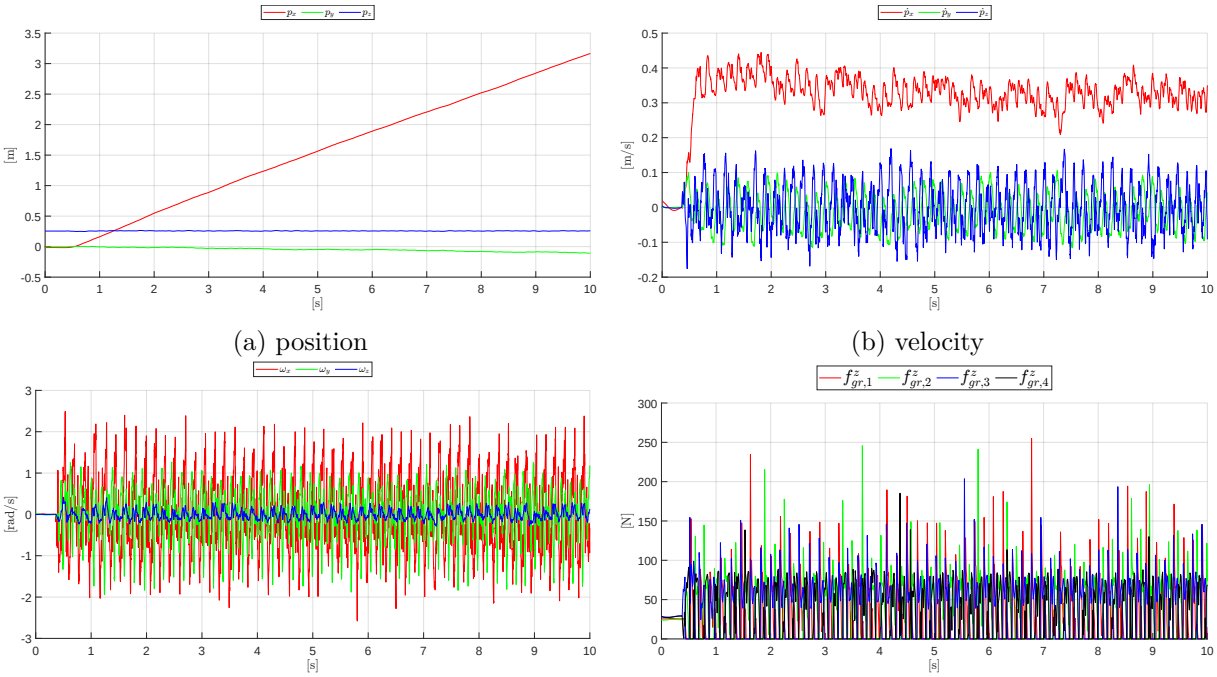
Figure 2.8: new bound, $v_{x,d} = 0.3m/s$



(a) position (b) velocity

(c) angular velocity (d) GRF_z

Figure 2.9: pace, $v_{x,d} = 0.3m/s$



(a) position

(b) velocity

(c) angular velocity

(d) GRF_z

Figure 2.10: gallop, $v_{x,d} = 0.3m/s$

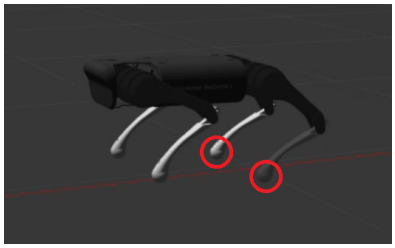


Figure 2.11: New bound gait

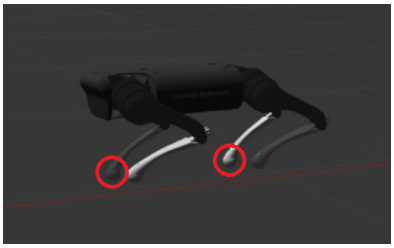


Figure 2.12: Pace gait

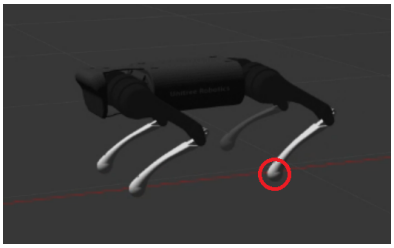


Figure 2.13: Gallop gait

2.3 RRT Path Planning

When the robot moves in an environment full of obstacles, a proper motion planning algorithm must be developed. In this work, it was decided to implement a Rapidly Exploring Random Tree (RRT) as global planner, while a simple DWA local planner was used as local planner. The DWA local planner is based on the idea of the Dynamic Window Approach (DWA) algorithm. This local planner is already available within the ROS environment, and in fact it can be considered the ROS standard for local planners, or at least one of the most widely used. However, regarding the RRT, its implementation is not directly available, so it was necessary to register it as a plugin with ROS Package System. To realize this, [13] and the example templates in [14] have been used as reference guidelines.

The first step that was carried out was a simple change of the world in which to spawn the robot. In [1], worlds with some obstacles in it are already available. However, for convenience, the world contained inside the rl_racefield package was chosen. Many tests have already been done in this map for other projects [11]-[12], so this world was preferred for conducting all the experiments. To do this, the rl_racefield package was added to the workspace. Then, the normal.launch file within the unitree_gazebo package was appropriately modified to call the rl_race_field.world. Additionally, it was necessary to change the robot's spawn position from

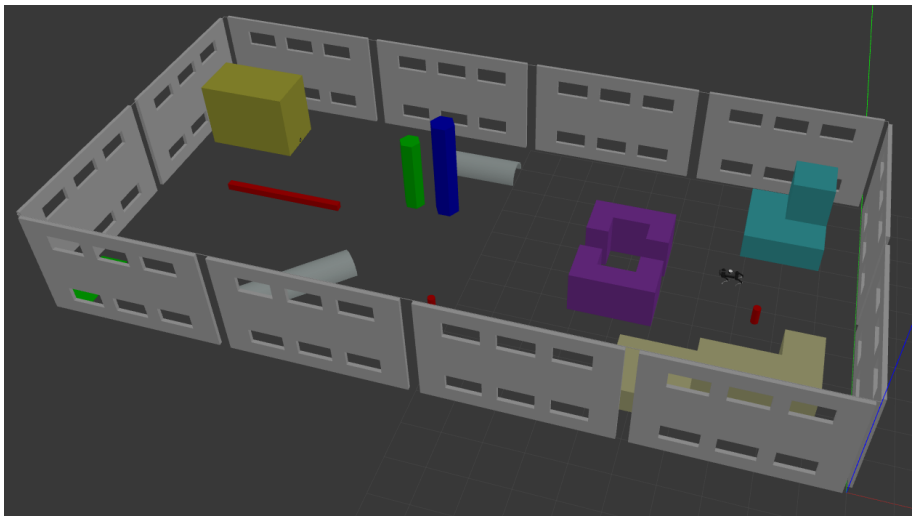


Figure 2.14: Chosen world

$x = 0, y = 0, z = 0.4, yaw = 0$ to $x = -3, y = 6, z = 0.4, yaw = 3.14$ both in the normal.launch file and in the qr_gazebo_controller_manager.cpp file (in which it is possible to reset the robot pose once the control is launched), that is in the quadruped package. In Fig 2.14, an image of the selected world is shown, with the robot already in the desired starting position.

The second step was the change of sensor. The A1 Robot provided by the reference repository was initially equipped with a Velodyne VLP-16 3D lidar sensor. However, the use of this sensor required a conversion from pointcloud to laser scan. This was quite computationally demanding, causing the control simulation to fail sometimes. This strategy was completely unnecessary for the following analysis, therefore, this sensor was replaced with a 2D Hokuyo Laser Scanner. For simplicity, the .xacro file referring to the sensor was not entirely modified, but only the part associated to the gazebo plugin. In other words, the new sensor will have the same textures, shapes, and reference frames of the VLP-16, but it will behave as a 2D Hokuyo.

Inside [1], only the use of gmapping is already implemented, and it does not directly include the use of path planning, as the robot has to be moved manually through the use of the keyboard. In this case, the goal was not to perform Simultaneous Localization and Mapping (SLAM), but rather to have the world map available to the use of the RRT, and then tackle only the problem of localization. Consequently, the already present implementation of gmapping was only used to build the world map. Specifically, the world was first opened with the command

```
$ roslaunch unitree_gazebo normal.launch rname:=a1 use_xacro:=
   true use_lidar:=true
```

Then, the control and gmapping were started with the command

```
$ rosrun demo demo_slam_gmapping
```

With simple keyboard commands, the robot was moved to every corner of the world in order to perform a complete mapping of the world. Once a satisfactory map was achieved, it was saved using the map_server package with the following terminal command

```
$ rosrun map_server map_saver
```

This command outputs two files: map.pgm and map.yaml. These two files were then placed inside a folder named maps, in the rl_racefield package. The file map.pgm is shown in Fig 2.15.

After this initial phase, it is possible to move on to the actual implementation of the path planner. What the demo_slam_gmapping node does is nothing more than launching

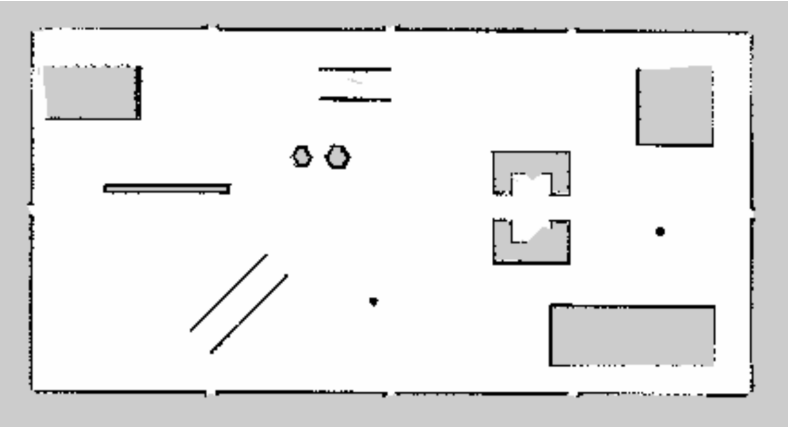


Figure 2.15: Map of the world

the gmapping.launch file, which handles the navigation part with gmapping, and also running the demo_publish_odom node, which manages the robot control part. Consequently, in order to modify only the planning part, the demo_publish_odom node was not modified, while the gmapping.launch file was replaced with a new launch file called nav.launch. This file has four main tasks

- Provide the map to move_base through the map_server node from the map_server package
- Start the move_base node, specifying which local and global planners to use
- Start the Adaptive Monte Carlo Localization (AMCL) node to localize the robot
- Activate the plugin related to the RRT global planner
- Launch RVIZ with a proper configuration to visualize the RRT planner

More specifically, besides assigning the desired local and global planners to move_base, it was also necessary to set all the parameters related to the global and local costmaps, as well as those related to the DWA planner, the RRT planner, and the general move_base parameters. All the corresponding .yaml files can be found in the slam package within the config/RRT folder. The name of the package has been kept as slam for simplicity, but it is important to note that the SLAM algorithm is not performed here. In fact, the main problem is only related to the localization of the robot now that the map is known. As mentioned previously, this problem was

Figure 2.16: Localization of the robot using AMCL node and 2D Hokuyo Laser Scanner

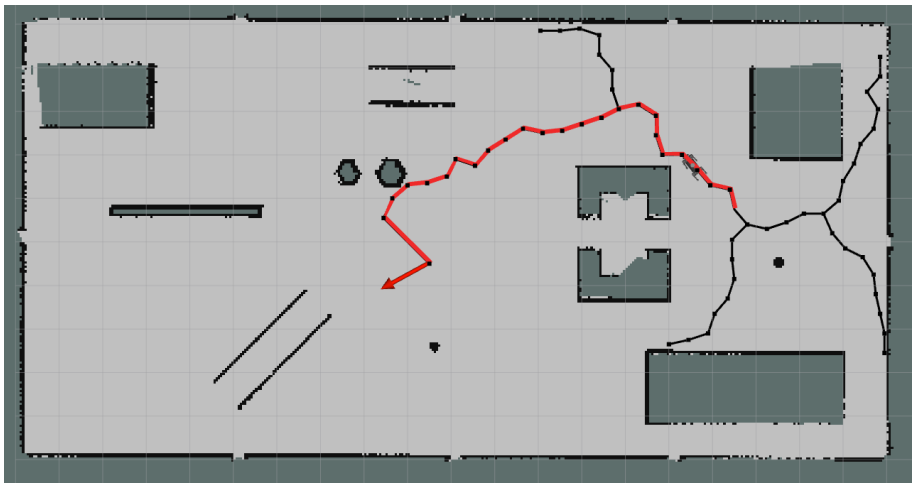


Figure 2.17: Example of usage of the RRT global planner

effectively solved using the AMCL node, which is an implementation of a particle filter based on laser scans. In Fig 2.16 it is shown the localization of the robot exploiting AMCL node and the new 2D Hokuyo Laser Scanner.

As mentioned before, the implementation of the RRT global planner requires to add a new planner in ROS as a plugin. To implement this, it's necessary to include the srv_client_plugin package, useful for registering the plugin with the ROS package system, that uses the pp_msgs message type. Then, three Python files have been expolited: RRT_code.py, RRT_server.py, treeviz.py. These were added to the slam package, in a folder called scripts. The file RRT_-code.py contains the main logic of the planner. The RRT algorthim has been implemented choosing to extract 50 random points q_{rand} at each loop, i.e. $max\_iterations = 50$. For each q_{rand} and the corresponding nearest point on the graph q_{near}, it has been used $\delta = 10$ in order to find q_{new}. Then, a collision check algorithm is exploited in order to check collisions between q_{near} and q_{new} choosing a sample step equal to 0.5. Afterwards, if it's not possible to find a path connecting q_{end} to the graph, the graph construction loop is repeated up to a maximum value set as $max\_loop = 100$. If this is reached without connect q_{end} to the graph, an error is reported.

The file RRT_server.py is necessary only to make this planner available to the move_base node, while the file treeviz.py is used for the graphical visualization of the tree and the final path in RVIZ. An example of a tree with the corresponding path is shown in Fig 2.17.

As shown in the respective videos, it is clear that the implementation of the RRT global planner was successfully carried out. In particular, once the goal is assigned with the simple RVIZ 2D Nav Goal command, a feasible path free of obstacles is quickly found (also because the map does not have unconnected areas or very narrow corridors). Then, the robot follows the indicated path, maintaining a correct localization thanks to the use of the particle filter, while keeping a quite moderate speed. As soon as the goal is reached (within a certain tolerance), this success is reported on the terminal, and the robot waits to receive a new goal to reach on the map.

Some remarks include the fact that in order to improve simulation performance, it was necessary to reduce the $< real\_time\_update\_rate >$ parameter (that defines the update frequency of the simulation) inside rl_race_field.world to 100, starting from a value of 1000. Without making this change, the computational demand was too heavy to obtain a satisfactory simulation. Moreover, the implementation of the planner could equivalently have been done in C++, and the overall structure of the code would have remained essentially the same.

Chapter 3

Data-driven Control

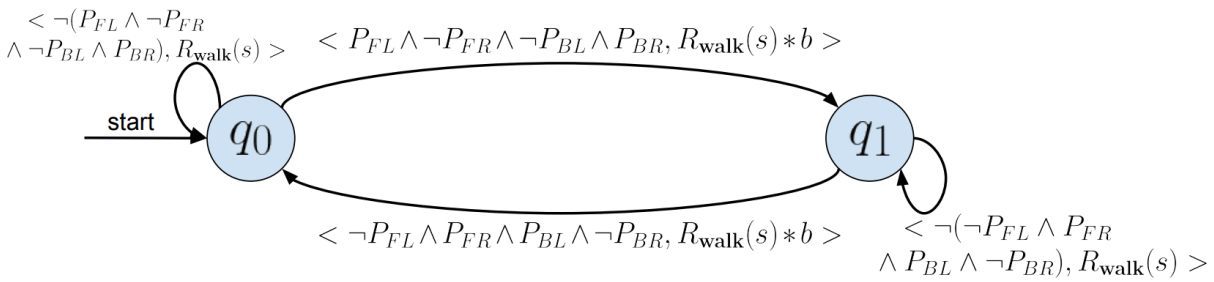
Model-based techniques, such as Model Predictive Control, have been widely used for controlling quadrupeds, demonstrating good effectiveness in a lot of cases. However, there are many scenarios in which model-based techniques fail to guarantee optimal performance. For example, on highly challenging terrains with a lot of irregularities, or on flat surfaces with very low friction coefficients, these controllers may be not the best choice. In such contexts, techniques based on Deep Reinforcement Learning (DRL) become crucial. These methods do not require precise knowledge of the system model and are capable of learning control policies directly from experiential data, adapting more flexibly and robustly to a wide range of situations.

This part of the project is an extension of [2]. Specifically, four main improvements have been made. First, the ANYmal B robot was replaced with a robot more similar to the Unitree A1, namely the Unitree Aliengo robot. This change required to re-tune the initial reward function, since, for example, the weights related to the joint torques and speeds were too low to be suitable for this robot. Additionally, the reward function was extended with more advanced terms in order to account for foot slip, foot height during locomotion, and the smoothness of the actions. The third extension involves training policies not to follow a single fixed speed and only move forward but to learn to move in all directions and at a generic desired speed within a certain range. Finally, the last extension involves the extraction of some quantities of interest from the simulator, such as the position, the linear and angular velocity of the robot, in order to make appropriate comparisons with the results obtained in Chapter 2 in ROS.

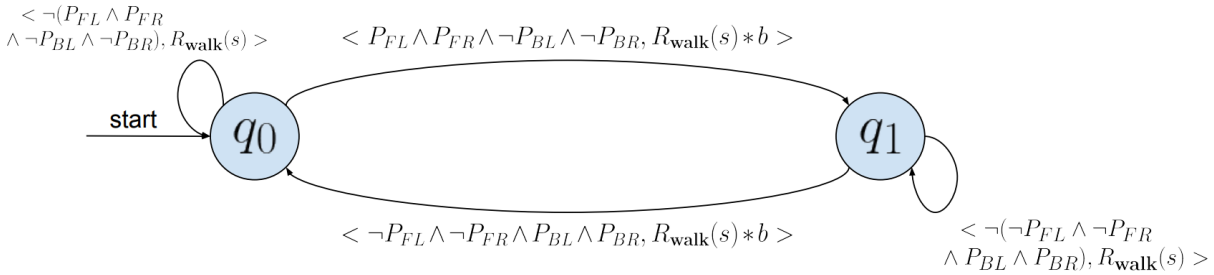
3.1 Reward Machines

Reinforcement learning algorithms assume that the environment is modeled as a Markov Decision Process (MDP) of the form $M = (S, A, T, R, \gamma)$. Here, S represents the state space, A is the set of possible actions that the agent can take, which may be either deterministic or stochastic, $T : S \times A \times S \to [0, 1]$ is the transition function that gives the probability of reaching state s' from state s after taking action a, $R : S \times A \times S \to \mathbb{R}$ is the reward received for taking action a in state s and transitioning to state s', and γ is the discount factor, indicating how much future rewards are valued compared to immediate rewards.

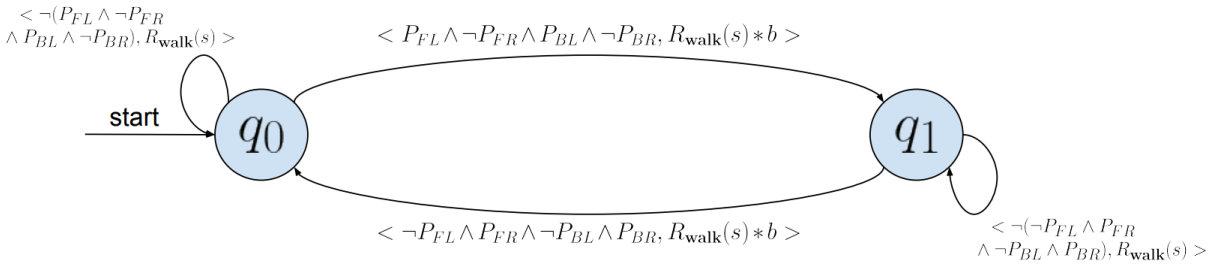
In Reinforcement Learning applications, Reward Machines (RMs) are used to manage milestone sub-goals in larger tasks [8]. Trying to train a quadruped to perform very specific gaits using only standard reward functions, which involve, for example, encoder information, linear and angular velocity, height, and orientation, is quite difficult. The reason is that the reward function is basically a black box for the robot. So, even if the reward function is accurately modeled, all weights are well tuned, and the algorithm is set up correctly, what the robot receives based on its actions is simply a scalar number. Therefore, following a process of exploration and exploitation interacting with the environment, the robot will generally learn to walk through a gait that is not easily predictable and can easily change from one training session to another.



(a) Trot gait



(b) Bound gait



(c) Pace gait

Figure 3.1: Reward Machines associated to the considered gaits

What is needed is simply a small guide, a sort of shortcut, that can allow the robot to learn to move with a predetermined gait without imposing too many constraints. For this purpose, Reward Machines are particularly suitable.

Reward Machines basically specify sub-goals using a finite state automaton. So, essentially, they extend the robot's state with variables belonging to a higher level of abstraction, allowing the robot to be guided to perform a certain task, in this case walking forward, in a particular way to further increase the reward. RM are formally defined as the tuple $(U, u_0, F, \delta_u, \delta_r)$, where:

- U is the set of automaton states,
- u_0 is the initial state,
- F is the set of accepting states,
- δ_u is the transition function,
- δ_r is the reward function.

Since RMs are based on the robot's past history, if the robot's state, denoted by S, were left as unchanged, the reward function would no longer be Markovian. As mentioned above, it is therefore necessary to extend the robot's state with the variables belonging to U. The new state of the robot will therefore be $S \times U$, and thus, the agent, in addition to relying on the normal reward function related to the task, will use δ_u and δ_r to learn a series of sub-goals.

Figure 3.2: Trot gait

Figure 3.3: Bound gait

Figure 3.4: Pace gait

It is worth remarking that RMs can generally be used in any RL-based application. Indeed, they can be used to facilitate the learning of a certain task by specifying sub-goals, and thus, learning the locomotion of a quadruped is just one of many applications where this technique can be exploited.

However, in this case, it is possible to introduce a variable $P = \{P_{FL}, P_{FR}, P_{BL}, P_{BR}\}$, where $p \in P$ is a Boolean variable indicating whether the front-left (FL), front-right (FR), back-left (BL), and back-right (BR) feet are in contact with the ground. This variable is essential because it is precisely thanks to P and its value that it is possible to transit from one state to another in the finite state automaton of the RM. For example, let's consider the case where we want the robot to learn a simple locomotion using the trot gait. In this case, the goal is a continuous alternation between the state $P = \{P_{FL} = 0, P_{FR} = 1, P_{BL} = 1, P_{BR} = 0\}$ denoted as q_0, and the state $P = \{P_{FL} = 1, P_{FR} = 0, P_{BL} = 0, P_{BR} = 1\}$ called q_1. It is then natural to think of a finite state automaton composed of the two states q_0 and q_1, and to stimulate the transition from one state to another by multiplying the normal reward function related to the walking, denoted as $R_{\text{walk}}(s)$, by a positive scalar b. Actually, since in the initial phase of the training the total reward was negative, it was preferred to simply add a positive scalar to $R_{\text{walk}}(s)$. So, the reward function δ_r is defined as

$$\delta_r(u_t, a) = \begin{cases} R_{\text{walk}}(s) + b & \text{if } \delta_u(u_t, a) \neq u_t \\ R_{\text{walk}}(s) & \text{otherwise} \end{cases} \tag{3.1}$$

The finite state automata associated to the trot, bound and pace gait are shown in Fig 3.1.

However, it can be noticed that with such a reward, the robot would tend to transition from one state to another faster and faster to maximize the reward. Such a condition is to be avoided, and this is why it is necessary to introduce the parameter δ. Here, δ represents the minimum time that must be spent in either of the two states q_0 or q_1 before a transition can occur. In this way, it's possible to decide the frequency with which to make the transition. Optionally, this parameter can be appropriately modified during the training phase so that the robot can learn to walk at different frequencies while maintaining the same speed, for example. For simplicity, this parameter will be considered fixed and always equal to 10. This 10 does not refer to actual time expressed in seconds but refers to the number of steps taken inside the simulation. So, as soon as the robot arrives in state q_0, 10 steps must pass before it can go to state q_1 and receive the extra reward, and vice versa.

3.2 State Space and Reward Function

The state space is defined as $S = (q, \dot{q}, z, \phi, v, \omega, q_{t-1}^{target}, q_{t-2}^{target}, v_d, u, \delta, P)$, where q and $\dot{q}$ denote the 12 joint angles and velocities, z is the body height, ϕ is the body orientation expressed as the Euler angles XYZ, v, ω are the base linear and angular velocity expressed in the body frame, q_{t-1}^{target} and q_{t-2}^{target} are the last two target for the joints, v_d is the desired velocity along the x, y directions with respect the body frame, u is the current RM state, δ is the number of time steps

Description	Equation	Weight
Linear Velocity x, y	$exp(-0.8\|\|v_{d,x,y} - v_{x,y}\|\|^2)$	10
Yaw	ψ^2	-10
Angular Velocity z	ω_z^2	-0.1
Angular Velocity x, y	$\|\|\omega_{x,y}\|\|^2$	-0.001
Joint Torques	$\|\|\tau\|\|^2$	-0.0001
Joint Positions	$\|\|q - q_{init}\|\|^2$	-0.005
Joint Velocities	$\|\|\dot{q}\|\|^2$	-0.0005
Hip Positions	$\|\|q_{hip} - q_{hip,init}\|\|^2$	-1
Foot Slip	$\sum_{i=1}^{4} P_i \|\|\dot{p}_{x,y,i}\|\|^2$	-0.1
Foot Height	$\sum_{i=1}^{4} (p_{z,i} - 0.1)^2 \|\|\dot{p}_{x,y,i}\|\|^{0.5}$	-1
Action smoothness 1	$\|\|q_t^{target} - q_{t-1}^{target}\|\|^2$	-0.5
Action smoothness 2	$\|\|q_t^{target} - 2q_{t-1}^{target} + q_{t-2}^{target}\|\|^2$	-0.5
RM transition	$b = 400$	1

Table 3.1: Reward Function for all gaits

since the previous RM state change, and P is the vector of four boolean variables that are 1 if the corresponding foot is touching the ground, 0 otherwise.

For what concern the action space, that is the low level controller, we have that the joints are torque controlled, and a simple PD controller is used

$$\tau = K_p(q_t^{target} - q_t) - K_d\dot{q}_t \tag{3.2}$$

Let's now consider the probably most important aspect when dealing with Reinforcement Learning, namely the reward function [9]. Here, as already mentioned earlier, the main objective was to train the quadruped to walk along a generic direction having both x and y-components different from zero. Specifically, the desired x velocity, y velocity and the yaw are variables belonging to the state of the robot, and they are continuously changed during the training. The velocities are varied in the range $[0, 1]$ m/s, while, for simplicity, the yaw angle is always maintained constant and equal to zero. This is performed using a cycle, Fig 3.5, in which every 5 iterations the references are changed. We start by asking a zero velocity, then we impose only a $v_{d,x} \neq 0$, then only a $v_{d,y} \neq 0$, then we set both $v_{d,x} \neq 0$ and $v_{d,y} \neq 0$, and finally we come back to $v_{d,x} = 0, v_{d,y} = 0$. By doing this, a single policy will be able to go in all direction, and with different velocities. Actually, although the maximum velocity that was shown during the training to the agent was 1 m/s, during the test it is able to generalize, and without difficulties it can track also higher velocities, even greater than 2 m/s. The terms used within the reward function are indicated in Table 3.1, and their respective weights are shown in the *Weight* column. It is worth remarking that the same reward function was used for all gaits. The only difference in the training regards the part associated to the reward machines, but the R_{walk} is kept always constant. Regarding the choices of the weights, they were firstly chosen in such a way to be coherent to the quantity they are referring to, in order to balance all the terms appropriately. For example, it was possible to assign greater weights to the penalties associated to the joint positions since these quantities will generally be quite small. On the other hand, the weights associated to the joint velocities were chosen to be smaller. Naturally, after these reasoning, a trial and error procedure was necessary at the end in order to achieve the desired behavior. Regarding the terminal condition, an episode ends simply when one of the four calf of the robot makes some type of contact, even with itself, not necessarily with the ground. In such a case, a quantity equal to 200 is subtracted from the overall reward.

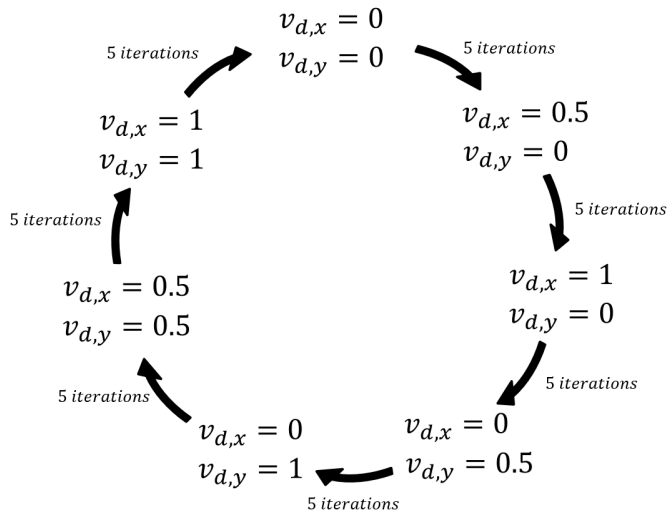


Figure 3.5: Cycle for learning multidirectional and variable-speed locomotion

3.3 Architecture and Gait Analysis

All the training have be carried out using the RaiSim simulator [15]-[16], that is a cross-platform multi-body physics engine for robotics and AI. The chosen algorithm was the Proximal Policy Optimization (PPO) [10], which can now be considered as the standard algorithm when dealing with continuous state and action spaces, or in general with very large and complex environments, such as those in most of robotic applications. This is an actor-critic algorithm, where two neural networks are trained in parallel. The actor learns the right action to take given a state, and the critic learns to judge the quality, i.e., the Q function of each input state. These two networks interact with each other and converge together towards a local minimum. For both networks, a multi-layer perceptron with three hidden layers with sizes of 256, 128, and 32 neurons was used, utilizing Leaky ReLU activation functions. The implementation available in the stable-baselines library was used for simplicity, and all the associated hyperparameters were left at their default values.

During each training episode, the policy network is run for 100 steps. Each episode lasts a maximum of 4 seconds, and, at each iteration, 30 different episodes are simulated in parallel to speed up the training. Another way to accelerate the training is to not graphically display the robot simulation on RaiSim at each iteration. In fact, to monitor the learning of the quadruped, stopping it if it was not considered of good quality, a graphical visualization every 25 iterations was considered enough. Regarding the sampling times, these were set to 0.0025 s for the simulation and 0.01 s for the control.

As mentioned earlier during the description of the action space, what the policy outputs given a certain state is basically the q^{target}. Then, the error between the q^{target} and the real q is computed, and the torques that will be applied to the robot are calculated with a simple PD controller. The value of the PD control gains has been kept to $K_p = 3$ and $K_d = 0.2$.

In Fig 3.6, the trend of the rewards mean for the three gaits considered in this project is shown. All three training were carried out more or less up to iteration 1200. From that point on, the reward tended to converge to a value of 56, and no significant improvement was achieved from that moment on. The trot and bound training were quite stable and also very similar to each other. In Fig 3.7, the relative loss function of the PPO algorithm during training is shown, and it is evident how the trend related to the trot and bound is extremely similar. The pace, on the other hand, exhibited much more unstable behavior. Even if these Deep Reinforcement Learning algorithms can sometimes not work perfectly, this result was not a case since the pace presented great instability in every training that was carried out, in which both the rewards

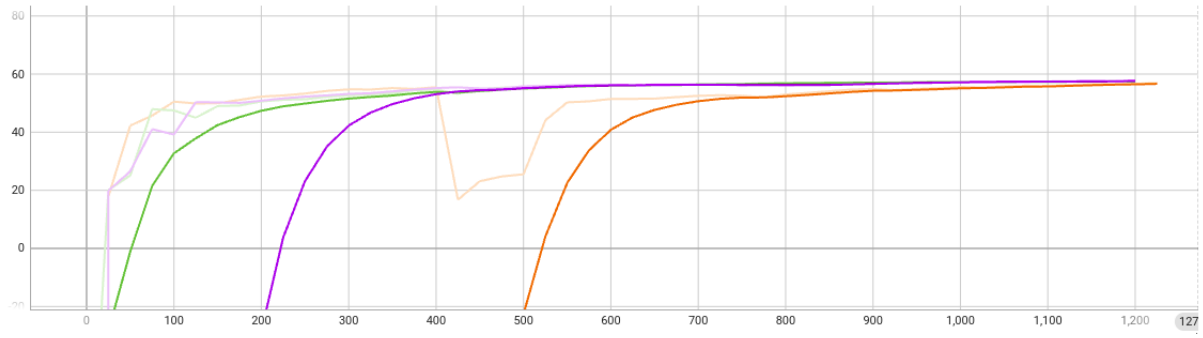


Figure 3.6: Reward mean evolution. Legend: violet: trot; green: bound; orange: pace.

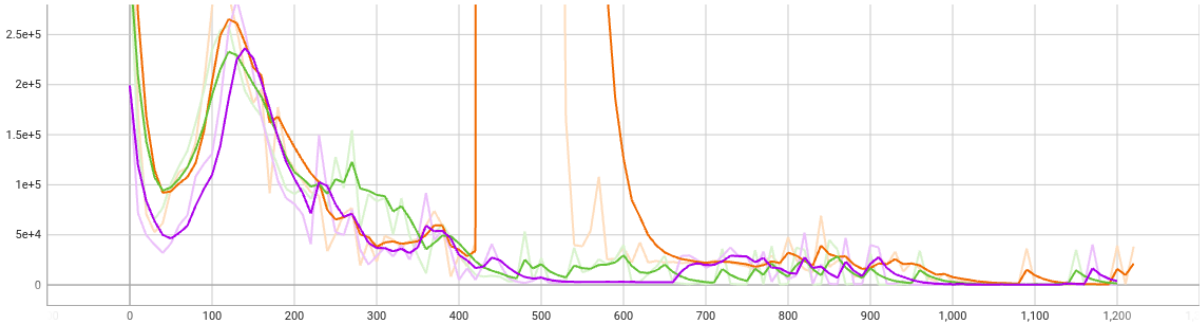


Figure 3.7: Loss function. Legend: violet: trot; green: bound; orange: pace.

associated to R_{walk} and those associated to the reward machine were slightly modified. In any case, 1200 iterations were still sufficient to allow the pace to converge to the local optimum.

Moreover, it is important to highlight that no curriculum learning techniques were applied during these training. In other words, penalties were not gradually added to the reward function to allow the robot to learn a simpler task first and then adjust it. Instead, they were considered in full from the beginning. This choice can sometimes cause the robot to neglect the positive reward related to walking and try to minimize all penalties simply by standing still. However, this phenomenon never occurred here thanks to the use of reward machines. In fact, during the first iterations of the training, the robot is not able to follow the desired speed but is still able to move its feet in place, and so it can quickly learn the desired gait. In this way, it's possible to inherently avoid the robot doing nothing, and as the iterations progress, it will gradually start learning to follow the desired speeds to further increase the reward.

Now let's move on to the analysis of the results. In Fig 3.8, 3.9, and 3.10, the position, linear velocities, and angular velocities of the three considered gaits are shown, in the particular case of a desired speed along x equal to 1 m/s, a lateral speed equal to zero, and a desired yaw angle always equal to zero.

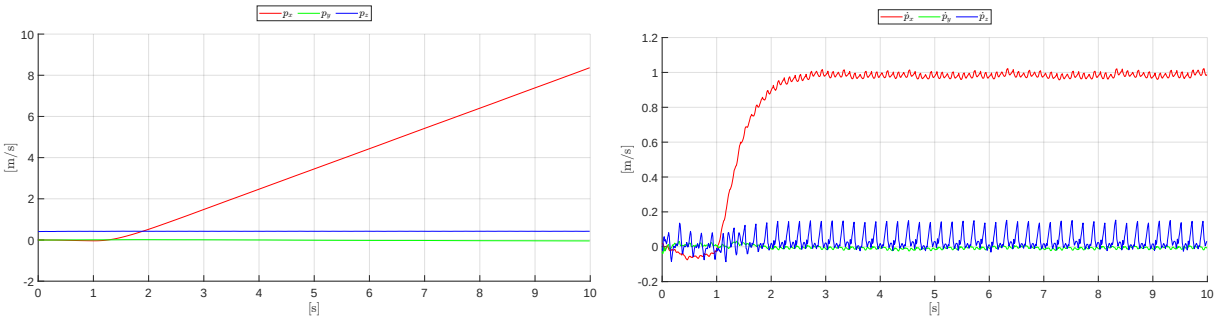
Regarding the trot gait, Fig 3.8, the position shows no significant drift along the y axis. In other words, v_y remains practically zero throughout the entire 10-second simulation. v_x correctly reaches the desired value of 1 m/s and remains quite stable. The most significant oscillations concern ω_x, but they are still quite contained, considering that the desired speed of 1 m/s for a robot like the Aliengo can already be considered very high. Qualitatively, the results are very comparable to those obtained with the A1 robot in Gazebo presented in the first chapter of the report.

Instead, moving on to the bound gait, Fig 3.9, it's possible to notice a slight drift along the y axis. After all, this gait is much more dynamic and energetic than the trot, so it is not surprising that it is more difficult for the policy to maintain a v_y actually at zero. Furthermore, we note that the desired speed of 1 m/s is not perfectly reached, since we have $v_x = 0.9$ m/s.

With reinforcement learning we are not actually computing at runtime an error between $v_{x,d}$ and v_x, but the robot's behavior depends on the choice of weights in the reward function during the training phase. Therefore, small offsets of this kind are completely normal. Finally, as was the case in the Gazebo simulations, the most important angular component is ω_y. In this case, the maximum values reached are slightly higher because the desired speed is greater than the case considered in ROS, which was 0.3 m/s.

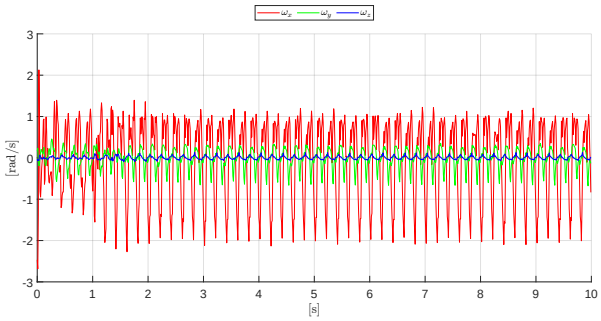
Regarding the pace gait, Fig 3.10, the considerations are similar. In this case too a slight lateral drift is present, although very contained. The desired speed is not perfectly reached, and in this case the highest oscillations occur on ω_x.

Having these policies available, each associated with a specific gait, it is now possible to exploit all of them even in a single task to use the one that best fits each situation. In the corresponding video on the attached github repository, an example of a simple task is shown where the robot first needs to move forward, then laterally, and then diagonally. In this specific case, it was decided to use the bound for the first part, the pace for the lateral translation, and the trot for the diagonal movement. In some cases, the combined use of different gaits might be the best strategy, and the same reasoning can similarly be applied to the model-based control.



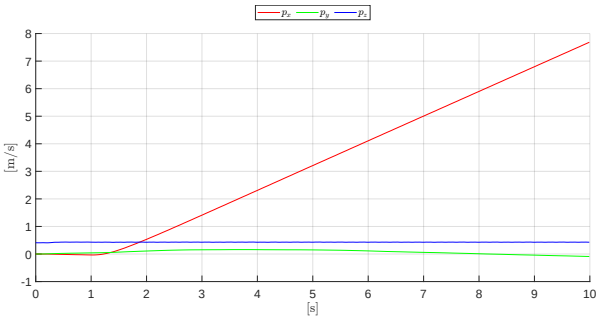
(a) position

(b) velocity

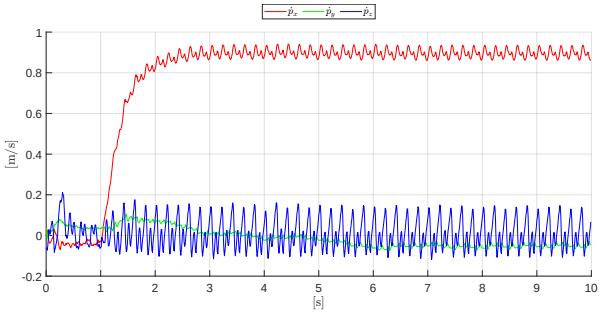


(c) angular velocity

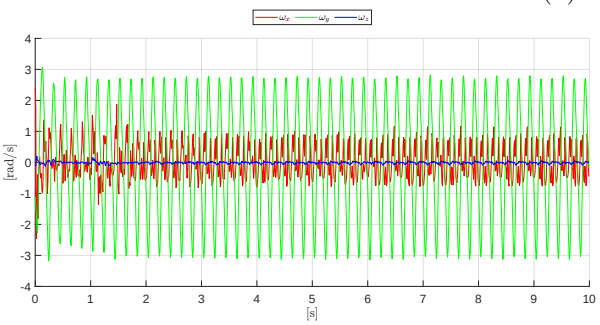
Figure 3.8: trot, $v_{x,d} = 1m/s$



(a) position

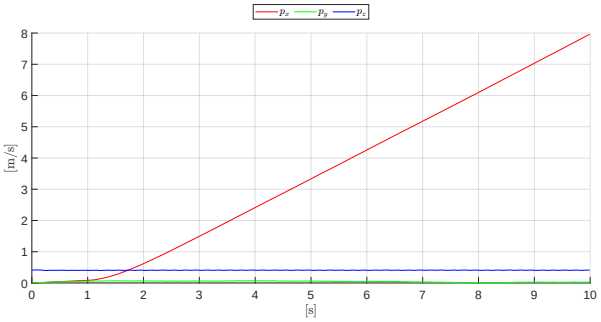


(b) velocity

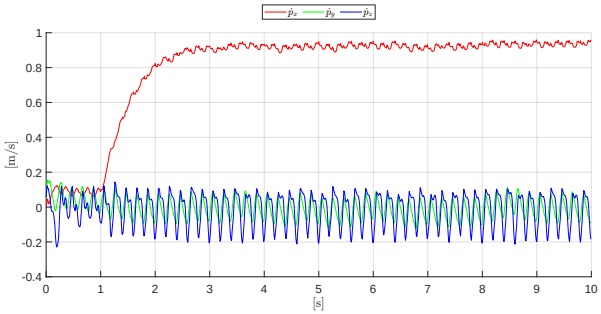


(c) angular velocity

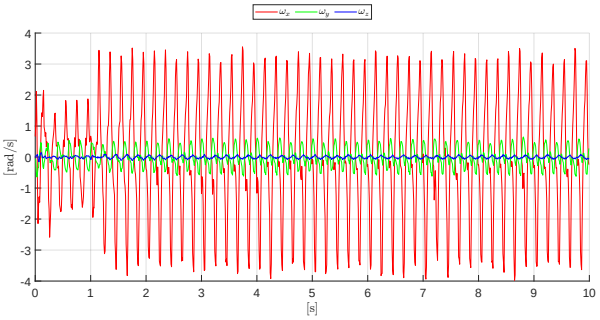
Figure 3.9: bound, $v_{x,d} = 1m/s$



(a) position



(b) velocity



(c) angular velocity

Figure 3.10: pace, $v_{x,d} = 1m/s$

Bibliography

[1] TopHillRobotics. "quadruped-robot". In: *https://github.com/TopHillRobotics/quadruped-robot/tree/main*.

[2] Emanuele Cuzzocrea. "Intelligent Robotics Project". In: *https://github.com/EmanueleCuzzocrea/Intelligent-Robotics*.

[3] Jared Di Carlo et al. "Dynamic locomotion in the mit cheetah 3 through convex model-predictive control". In: *2018 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2018, pp. 1–9.

[4] Gerardo Bledt et al. "Mit cheetah 3: Design and control of a robust, dynamic quadruped robot. In 2018 IEEE". In: *RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 2245–2252.

[5] Fabio Ruggiero. *Field and Service Robotics lecture slides*. Università di Napoli Federico II. 2024.

[6] Viviana Morlando, Ainoor Teimoorzadeh, and Fabio Ruggiero. "Whole-body control with disturbance rejection through a momentum-based observer for quadruped robots". In: *Mechanism and Machine Theory* 164 (2021), p. 104412.

[7] Donghyun Kim et al. "Highly dynamic quadruped locomotion via whole-body impulse control and model predictive control". In: *arXiv preprint arXiv:1909.06586* (2019).

[8] David DeFazio, Yohei Hayamizu, and Shiqi Zhang. "Learning quadruped locomotion policies using logical rules". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 34. 2024, pp. 142–150.

[9] Michel Aractingi et al. "Controlling the Solo12 quadruped robot with deep reinforcement learning". In: *scientific Reports* 13.1 (2023), p. 11945.

[10] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[11] Vito Daniele Perfetta. "mobileRobotControl". In: *https://github.com/danperf/mobileRobotControl.git*.

[12] EmanueleCuzzocrea. "Homework4". In: *https://github.com/EmanueleCuzzocrea/Homework4.git*.

[13] wiki.ros.org. "Writing A Global Path Planner As Plugin in ROS". In: *https://wiki.ros.org/navigation/Tutorials*.

[14] Roberto Zegers R. "Path planning intro". In: *https://github.com/rfzeg/path_planning_intro*.

[15] RaiSim. In: *https://raisim.com*.

[16] Jemin Hwangbo. "raisimLib". In: *https://github.com/raisimTech/raisimLib*.