DIETI. UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Control and Simulation of a Quadruped Robot in ROS/ Gazebo

Robotics Lab and Field Service Robotics Modules
Technical Report

Candidate

Alessandro Sofia - P38/08

Professor

Prof. Fabio Ruggiero
Prof. Johnatan Cacace

Index

Introduction

The simulation is run in Ubuntu 20.04 OS, using ROS Noetic as middleware and Gazebo 11.5.1 simulator. Matlab r2021a has been used for plotting the values encapsulated in the rosbag files extracted from the simulations. The code is entirely written in C++ using VisualStudio as editor. To build the package the option -DCMAKE_BUILD_TYPE is set to Release as it speeds up execution time for the optimisation problem and Towr computations.

Recorded video, available in appendix, demonstrate robot's capabilities. It is relative to the exploration of all the rooms in the sequence: Start,Room2, Room1, Room3,Start. Both the local minimum routine and the exploration routine, described in the following chapters, are shown.

Project Work Specification

This Project Work aims at implementing the control of a quadruped robot, here on referred to also as DogBot, developing a navigation algorithm for exploration through an unknown environment.

The task the DogBot has to perform is to retrieve a specific QR Code Marker placed in one of the rooms, fixed on a wall in an unknown position.

The robot starts in a known position, then requests to a server the coordinates of the first room to explore. After receiving the coordinates of the room, the DogBot navigates the unknown environment to reach it. When arrives to the goal, it starts to explore the room searching for the hidden QR Code. All rooms have a hidden QR Code so the robot continues to explore the room until one is found. If the one found does not contain the ID it is looking for, then navigates to the next room, otherwise returns to the initial spawn position. At the end of the exploration a file containing the map of the navigated environment is produced.

The localisation is achieved by directly extracting all the necessaries values from the gazebo simulation. In the navigation path there are obstacles the robot has to avoid. Path planning is carried out with an online version of the artificial potentials method, and the local minimum condition is managed by generating random goal values.

World Description

The environment chosen for the final simulation is a 8x14m rectangle, with walls defining 3 rooms, an open space, and a starting position. Construction cones are randomly placed as obstacles crowding the open space as by project work requirements. In each room there's a QR Code encrypting a unique ID. World dimensions are suitably defined to enable the simulation to finish in a reasonable amount of time, while demonstrating all the capabilities of the robot.

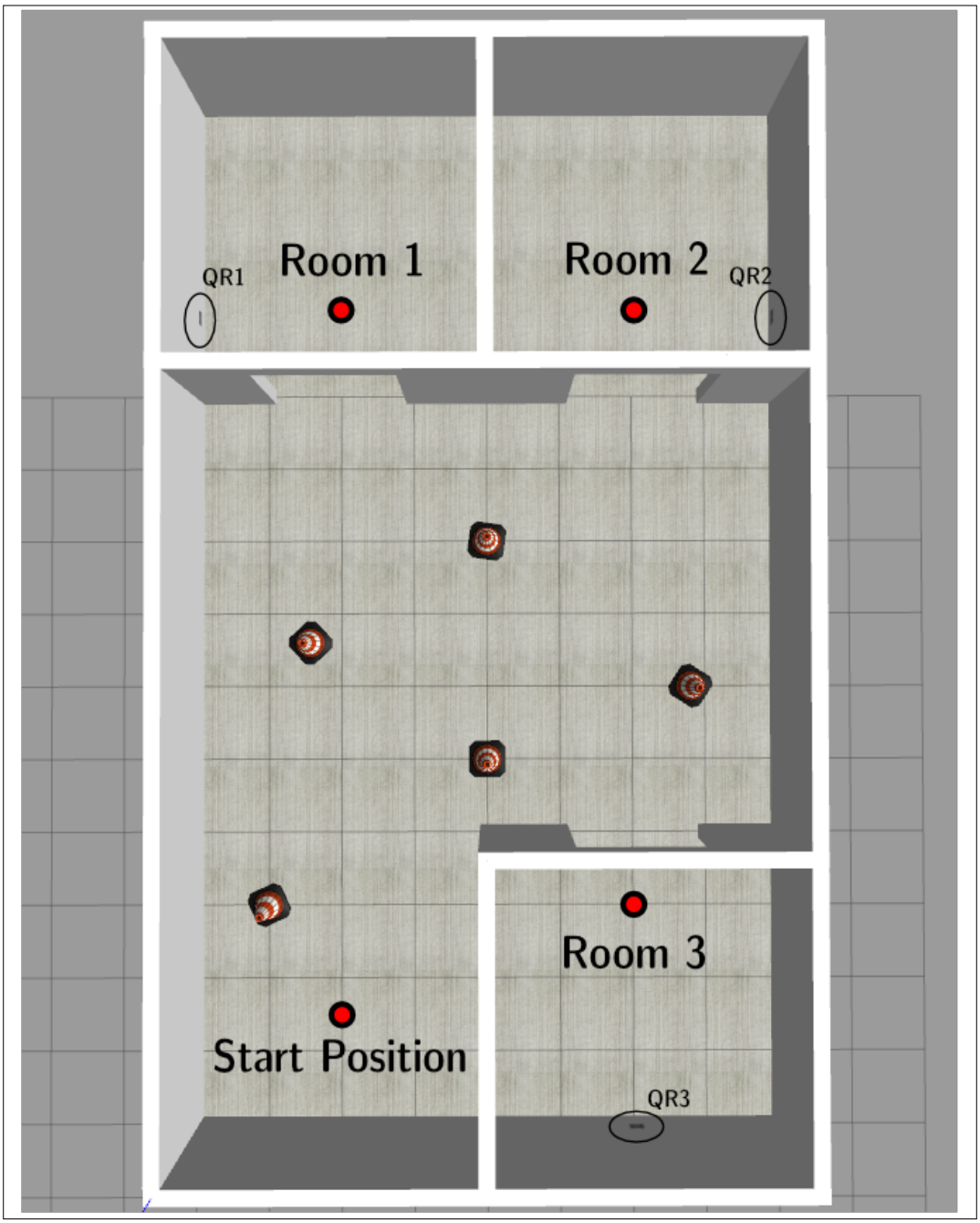


Figure 1 —World Screenshot

Coordinates describing the world used in the simulations are the followings.

World bottom left corner is the world origin (0,0).
Robot's starting position is in the open space at (2,2).

Rooms coordinates are chosen 1m from the entry door.
Room1 - (2,11)
Room2 - (6,11)
Room3 - (6,3)

Each QR Code is placed at 0.5m from the ground and on different relative positions in each room, in order to extensively test the exploration routine.

The QR Codes encoded ID's are the following:
QR1 - ID1234
QR2 - ID0101
QR3 - ID2997

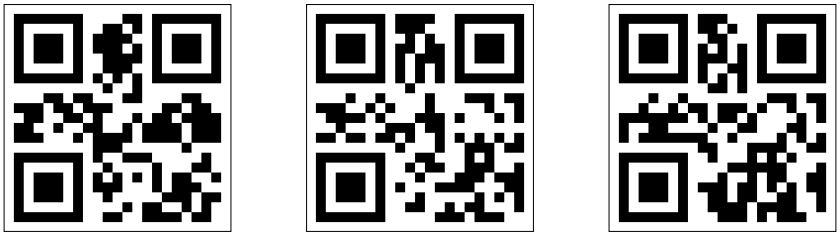


Figure 2.1, 2.2, 2.3 — QR Markers

QR Code dimensions are 0.2x0.2x0.001m
Construction Cones have a squared base 0.5x0.5m and max height is 1m.
Each room is a $16m^2$ square.
Walls thickness is 0.2m.

To perceive World's dimensions in reference with obstacles and the robot, it can be observed that each square in the image is 1x1m.

Robot Description

Figure 3 — DogBot in the sim environment

The quadruped robot used for the project work is the DogBotV4 by ReactRobotics [1]. The robot has four legs, each one equipped with three actuated revolute joints: a "Roll Joint", a "Knee Joint" and a "Pitch Joint".

The first two connect the robot body to the upper leg, while the latter connects the upper leg to the lower part. The first joint axis is parallel to the robots' body longitudinal axis, while the second and third joints have both the axis normal to the plane of the leg. While the first joint is used to enable lateral movements, the other two are used to lift the leg off the ground.

The legs configuration chosen is the elbows configuration (i.e. all the legs point backwards) which facilitate the push-off impulse. The DogBot mass is 21kg, about 12kg for the body and 2kg for each leg. The body length is about 0.9m while the upper and lower leg are 0.3m each.

The URDF used for the simulation is a modified version of the one available on GitHub [1]. The robot sensor added on the xacro file are:
- Contact Sensors
- Camera Sensor
- Lidar Sensor

Contact sensors are placed on each foot, the data extracted enables to recognise if a foot is in contact with the ground and the contact force.

The Camera and Lidar sensor are attached on the uppers side of the body link, the former shifted towards the heading direction of 0.15 m, the latter of 0.3 m.

Camera Specs:

Horizontal fov: 1.39
Img width: 1280
Img height: 720
Format: R8G8B8

Lidar Specs:

Samples: 360
Min/Max angle: -3.14/3.14
Min range: 0.4
Max range: 2.5

Project Skeleton Description

The nodes developed for the package are five. A position server, a path planner node, an odometry node, a controller node, an image scanner node. All the nodes tasks are explained in the next chapters.

In the following image a summary scheme represents all the nodes composing the package, complete with topics published and message types.

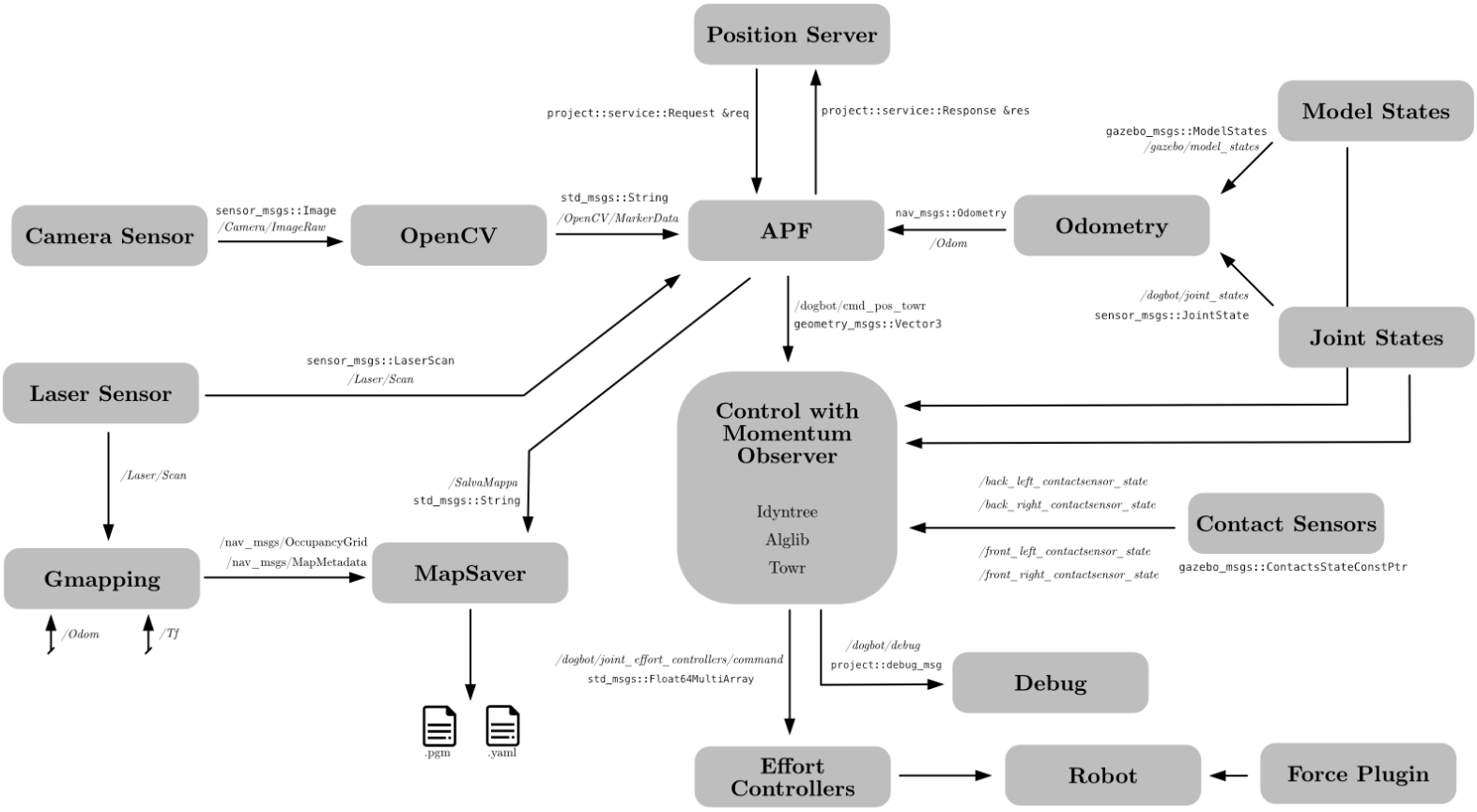


Figure 4 — Package Summary Scheme

Control System

Using the control scheme described in [2] a wrench-based optimisation problem is defined, using the same notation. Let the control variable $\zeta = [\ddot{r}_c^T \quad \ddot{q}^T \quad f_{gr}^T]$, the quadratic problem is defined as:

$$\min_{\zeta} f(\zeta)$$

$$s.a. \qquad A\zeta = b$$
$$D\zeta \leq c$$

The cost function defined in [2] is $f(\zeta) = \|J_{st,com}^T \Sigma\zeta + J_{st,com}^T \hat{f}_{st} - w_{com,des}\|_Q + \|\zeta\|_R$, although to define it in alglib the following manipulation have been made.

The qp problem in alglib needs to be defined as: $\min_{\zeta} \frac{1}{2}\zeta^T Q\zeta + c^T\zeta$.

Therefore considering the equivalence :

$$\frac{1}{2}\|T\alpha - p_r\|_w^2 = \frac{1}{2}\alpha^T T^T W T\alpha - p_r^T W T\alpha + \frac{1}{2}p_r^T W p_r$$

and choosing $T = J_{st,com}^T\Sigma$, $\alpha = \zeta$, $p_r = -J_{st,com}^T\hat{f}_{st} + w_{com,des}$

The chosen cost function is: $f(\zeta) = \zeta^T(T^T W T + R)\zeta - T^T W^T(w_{com,des} - J_{st,com}^T\hat{f})\zeta$

$$T = J_{st,com}^T\Sigma \qquad \Sigma = -\begin{bmatrix} 0_{6+(4-n_{st})x6+(4-n_{st})} \\ I_{3n_{st}x3n_{st}} \end{bmatrix} \qquad W = \begin{bmatrix} 50 & & \\ & \ddots & \\ & & 50 \end{bmatrix}_{6x6}$$

$$Q = -T^T W T + R \qquad c^T = w_{com,des} - J_{st,com}^T\hat{f} \qquad R = I_{30x30}$$

Equality constraints are defined as:

$$A = \begin{bmatrix} M_{com}(q) & 0_{6x12} & -J_{st,com}^T(q) \\ J_{st,com}^T(q) & J_{st,j}^T(q) & 0_{3n_{st}x3n_{st}} \end{bmatrix}$$

$$b = -\begin{bmatrix} mg \\ \dot{J}_{st,com}(q,\dot{q})v \end{bmatrix}$$

Inequality constraints are defined as:

$$D = \begin{bmatrix} 0_{4n_{st}x6} & 0_{4n_{st}x12} & D_{fr} \\ 0_{12x6} & M_q(q) & -J_{st,j}^T \\ 0_{12x6} & -M_q(q) & J_{st,j}^T \\ J_{sw,com} & J_{sw,j} & 0_{3(4-n_{st})x3n_{st}} \\ -J_{sw,com} & -J_{sw,j} & 0_{3(4-n_{st})x3n_{st}} \end{bmatrix}$$

$$c = \begin{bmatrix} 0_{4n_{st}} \\ \tau_{max} - C_q(q,v)\dot{q} \\ -\tau_{min} + C_q(q,v)\dot{q} \\ \ddot{x}_{sw,cmd} + \dot{J}_{sw,com}(q,\dot{q})\dot{r}_c + \dot{J}_{sw,j}(q,\dot{q})\dot{q} \\ -\ddot{x}_{sw,cmd} - \dot{J}_{sw,com}(q,\dot{q})\dot{r}_c - \dot{J}_{sw,j}(q,\dot{q})\dot{q} \end{bmatrix}$$

Chosen characteristic values:
- to avoid slippage $\mu = 0.5$ for the friction cones definition in the D_{fr} matrix
- to limit control action $\tau_{max} = -\tau_{min} = 60N$

To take into account inaccurate model parameters and external forces is also implemented the momentum-based observer as described in [2]. The estimator degree chosen is 1. So γ_1 is computed as :

$$\gamma_1(t) = K\left(\rho(t) - \int_0^t (\hat{F}(\sigma) + \alpha(\sigma)) \quad d\sigma\right)$$

where:

$\hat{F} = \gamma_1$
$K = I_{12x12}$
$\rho = M_q\dot{q}$
$\alpha(\sigma) = C_q^T\dot{q} + \tau + J_{st,j}^T f_{gr}$

In the code implementation the integral is discretised and $\hat{F}(\sigma)$ is referred to the estimation obtained at the previous time-step. In order to obtain the estimation of the force at the feet the transform using the Jacobian matrix is needed: $\hat{f} = J_j^{T^\dagger}\hat{F}$.

As it is assumed $\rho(0) = \gamma_1(0) = 0$ the estimator kicks off before the first control loop. The value needed by the observer are the ground reaction forces, obtained by reading the published messages by the contact sensors, the leg's input torques $\tau^\star$ and joints vector position and velocity.

The controllers are defined in the control.yaml file in the /conf directory. Controller type for all the joints is effort_controller, and to send a simplified message for the control it is used a JointGroupEffortController.

Libraries used for the node

ALGLIB

The quadratic optimisation problem is solved using ALGLIB [3], so a minqp object is defined following these steps:

1. It is created an instance of the optimiser with *minqpcreate*
2. Quadratic term is set with *minqpsetquadraticterm*
3. Linear term is set with *minqpsetlinearterm*
4. Linear constraints are set with *minqpsetlc*
5. Set the algorithm and stop criteria with *minqpsetalgodenseaul*

To set linear constraints at step 4. it is also necessary to define a new matrix "Lt" containing 0 for each equality constraint, -1 for inequality ones.

In fact two different quadratic problems are defined, one for the stance and one for the swing motion. The first problem dimension are fixed as the number of legs in contact with the ground is always set to 4. The number of control variables is 30, while the number of constraint is 58. The second problem is defined with a dynamic constraint matrix to implement different gate types and step motions.

IDYNTREE

iDynTree is a library of robots dynamics algorithms for control, estimation and simulation, specifically designed for free-floating robots. [7]

Two main iDynTree classes are useful for the project, the first one is iDynTree::Model used to represent the quadruped model, then iDynTree::KinDynComputations used in order to compute forward kinematics and dynamics quantities.

Relevant issues: randomly during the simulation iDynTree gives empty values for the base pose, in order to avoid spikes in the error reference before each cycle is carried out a check on this occurrence.

TOWR

The library used to generate the trajectories for the centre of mass (CoM) and the legs is Towr. [4] The trajectory optimisation is based on the solution of a Nonlinear Programming Problem (NLP) solved using Ipopt through the library interface Ifopt [5][6].

To add the DogBot as a new model, the library is extended by defining the Kinematic and Dynamic models into a new ".model" file in the /model/example towr folder. Values for the Single rigid body dynamic model and the nominal end effectors stance were extracted from the DogBotV4 URDF file.

To define the NLP problem we first define which model to use and the specific terrain, then CoM and legs initial linear and angular position, velocities and accelerations are updated from iDynTree. Desired CoM pose is sent by the trajectory planner node. The chosen gate type is the trot, and the total phase duration is set to 1s. The solution is then obtained setting costs and constraints for the NLP problem, and choosing a finite difference method to solve it through Ipopt.

Navigation

The navigation stack is composed by four parts. A Server which task is to answer the APF node with the rooms' coordinates, the Artificial Potential Field Planner (APF) node needed for generating the path, the Gmapping and Map Server nodes used to save the map.

Server

A server is implemented to store and send the rooms' position to the planner, when a room number is required. The planner starts with the variable "room_explored" set to 1, then acting as a client makes a call to the server asking for the coordinates of the new room. Each time a room is explored the value of room_explored is incremented by one. When the value of room_explored reaches a number greater than 3 the server sends the starting position to the client. The value of room_explored is set to a value greater than 3 also when the requested QR Code is found.

The custom service message used is composed as follows:

```
int32 in
---
float64 goal_x
float64 goal_y
```

So client needs to send an integer value containing the desired room number. Server answers with the room coordinates sent as two double values.

Artificial Potentials

As specified in the project work and due to fact the environment is unknown, the navigation is achieved by implementing an online version of the artificial potentials method. To build a force an attractive potential to the goal is combined with a repulsive potential computed by taking into account each obstacle spotted by the lidar sensor. So the first part of the algorithm needs to detect the number and position of the obstacles and compute a force for each one.

To achieve this behaviour it is implemented a check on the number of adjacent laser rays which distance is lower then a threshold — range of influence $\eta_{o,i} = 2.2m$ — in doing so we are also checking for the end an obstacle and computing the force.

The repulsive force for each obstacle is then defined as follows:

$$f_{r,i} = -\nabla U_{r,i}(q) = \begin{cases} \frac{k_{r,i}}{\eta_i^2(q)}\left(\frac{1}{\eta_i(q)} - \frac{1}{\eta_{o,i}}\right)^{\gamma-1}\nabla\eta_i(q) & , \eta_i(q) \leq \eta_{o,i} \\ 0 \quad , \quad \eta_i(q) > \eta_{o,i} \end{cases}$$

where $\eta_i(q) = \min_{q' \in CO_i} \|q - q'\| = obstacle \quad distance.$

To compute the gradient it is used the following approach, let the distance in 2D be described as:

$d = \sqrt{(x_f - x_i)^2 + (y_f - y_i)^2}$, with $(x_i, y_i) = (0,0)$ as it is the starting point of the lidar beam.

We can compute the gradient as $\nabla(d) = \left(\frac{x}{\sqrt{x^2+y^2}}\hat{x} + \frac{y}{\sqrt{x^2+y^2}}\hat{y}\right)$ and substituting the obstacle position.

This is retrieved by applying the following: $\begin{bmatrix} x = \eta_d \cos(\theta + yaw - \frac{\pi}{2}) \\ y = \eta_d sin(\theta + yaw - \frac{\pi}{2}) \end{bmatrix}$,

with:

- $\eta_d =$ obstacle distance from the robot's lidar
- $\theta = n_{laser} \cdot l_{increment}$ describing the angle between laser's first ray and the obstacle ray
- $yaw =$ current robot's yaw in the world frame.

It must also be subtracted $\frac{\pi}{2}$ because of the rotation between the robot's reference frame and world reference frame.

Then an error is then computed as the difference between current and goal position.

Applying the force: $f_{att} = \begin{cases} f_a = -\nabla U_a = K_a e(q) \\ f_b = -\nabla U_b = K_a \frac{e(q)}{\|e(q)\|} \end{cases}$ the DogBot linearly approaches the desired position.

The use of two different attractive potential is needed to limit the control action value when the goal position is far away from the starting position. As by theory the transition is carried out when $\|e(q)\| = 1$ to avoid spikes in the control action.

The total force is then computed as: $f_{tot} = f_{att} + \sum_i f_{r,i}$

Characteristic chosen values are:
$\gamma = 2$
$K_a = 10000$
$K_r = 2000$

As the total force is seen as an acceleration vector, it is twice integrated to obtain a position reference. It also computed the yaw reference using the atan2 function. To ensure that towr doesn't get a step greater than 0.15 m a cap on the maximum linear velocity is set. Also the robot's heading direction has a cap on the maximum variation of 0,2 rad. When the yaw reference error is more then 0,2 rad it runs a routine that makes the dogbot rotate on the spot.

To escape the local minimum problem, a routine checks for a condition in which the control force is lower than a set value, while the error norm is greater than 1.0 m. When this condition is true the DogBot steps back from the obstacles and then a new random position is passed as goal. The robot keeps following the new objective for 120 seconds or until the position is reached. Then the original goal is set again to try and reach it. For simulation purposes the random solutions are generated with an x position opposed to the one the robot wants to reach and a y position guiding it towards the middle of the world. With this approach the robot in various scenarios is able to exit rooms and avoid minima in few iterations.

Relevant additional issues :
1 — Circular Array Problem: Lidar values are stored in a classic array, while an object can extend between the first and last ray of the lidar, so to avoid an error computing it as two different objects there's a first routine to shift these values until the first ray is empty.

2 — Corner Errors: To define an obstacles the routine checks for the number of subsequent rays with a value minor then a set threshold, this means that two walls forming a corner are seen as one obstacle. During the navigation this leads to the computation of a single repulsive force that makes the robot oscillates from the left to the right wall without entering the local minimum threshold. To avoid this problem it is implemented a routine to consider two walls forming a corner as two

different objects. This check enables to compute more accurately the repulsive force and detect local minima conditions. The check works as follows: when an obstacle is seen for more than 120°, it is scanned for the condition in which there's a distance value of a ray which is a local maximum (i.e. the next and the previous rays distances are lower). If this condition is met, this ray, which is the "corner ray" distance is set to +inf, meaning that the routine that computes the repulsive force sees the walls as two different obstacles.

3 — Yaw Reference Error: using the atan2 function to compute the yaw desired direction, starting from the x and y goal position components, when crossing 3.14rad the reference value becomes -3.14rad. This behaviour is a problem while tracking the yaw reference making the robot try a 360° spin on the spot. To solve this problem a counter value is increased (or decreased) each time two subsequent reference values' sign changes. This condition is not met when values pass by zero but only near 3.14. Yaw value is therefore computed adding $2\pi * counter$ to the atan2 function. This yaw modified value is used in two cases, the first is to generate the rotation matrix needed to compute $w_{com,des}$, the second time to pass the yaw reference to the path generation routine (towr).

Data needed for the node are retrieved by subscribing to the following topics:

- "/odom" — message contains odometry info
- "/opencv/maker_data" — message contains QR marker decrypted string
- "/laser/scan" — message contains laser sensor data

Values generated by the node are published on the following topics:

- "/dogbot/cmd_pos_towr" — message contains next goal data for towr
- "/salvamappa" — message contains a string to save the map

Gmapping

The ROS node slam_gmapping provides laser-based SLAM (Simultaneous Localization and Mapping) used for the environment map generation. Using SLAM technique this node creates a 2-D occupancy grid map using the odometry and laser data. Data are retrieved by subscribing to the "laser/scan" topic. In order to combine laser data with the odometry this node also needs the transform tree data obtained by subscribing to "/tf".

Slam_gmapping publishes the occupancy grid and associated metadata on the following topics: "nav_msgs/OccupancyGrid" and "nav_msgs/MapMetaData".

To check all the transformations between frames the tf tree is saved as png using the rqt_tf_tree ROS package.

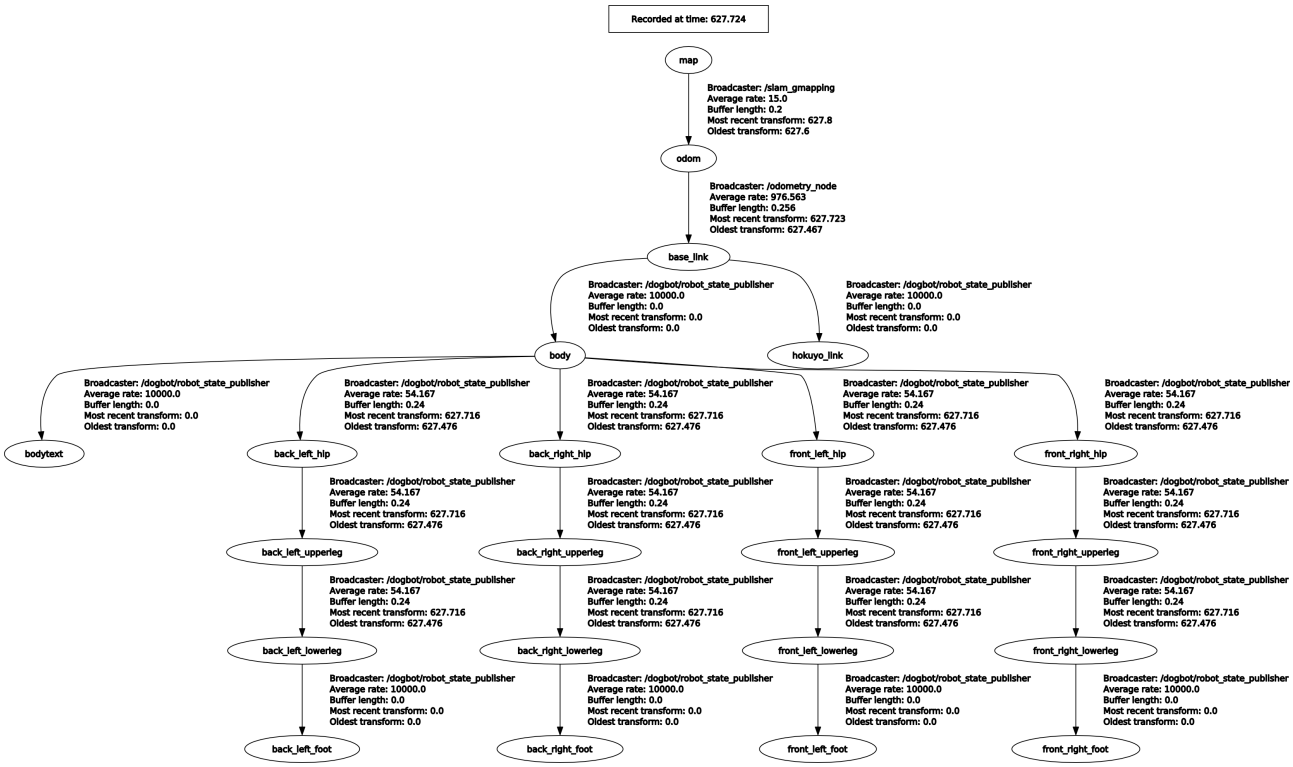


Figure 5 — Tf Tree

Exploration Routine

Once the robot has reached the room coordinates, sets a boolean value to begin the exploration routine and looks for the QR Code. The algorithm to explore the room works as follows. The DogBot walks in a straight line as long as the first ray of the LIDAR Sensor returns a value of >1.2m, which means the robot reaches a frontal distance with the wall of at least 1.2m. Then walks on his right side while looking at the wall until he reaches a lateral distance of 1.2m with the lateral wall. When he reaches this position, i.e. a corner of the room, turns at 90° degrees and repeats it until all the walls are explored and the QR Code found. Thus enables the robot to walk along the walls of the room. The algorithm stores the number of walls it has explored (by incrementing a counter when a rotation of 90° is completed) so knows when the DogBot faces the wall with the door, so it doesn't try to exit during the exploration. The hypothesis on which the algorithm is based is that the robot enters the room with a yaw almost orthogonal to the frontal wall and that inside the room there are no obstacles. If we remove the hypothesis the algorithm can be modified to randomly explore the walls by generating random values for the degree of rotations. Below a figure explains the routine, in blue the wall explored, in black the path followed by the robot.

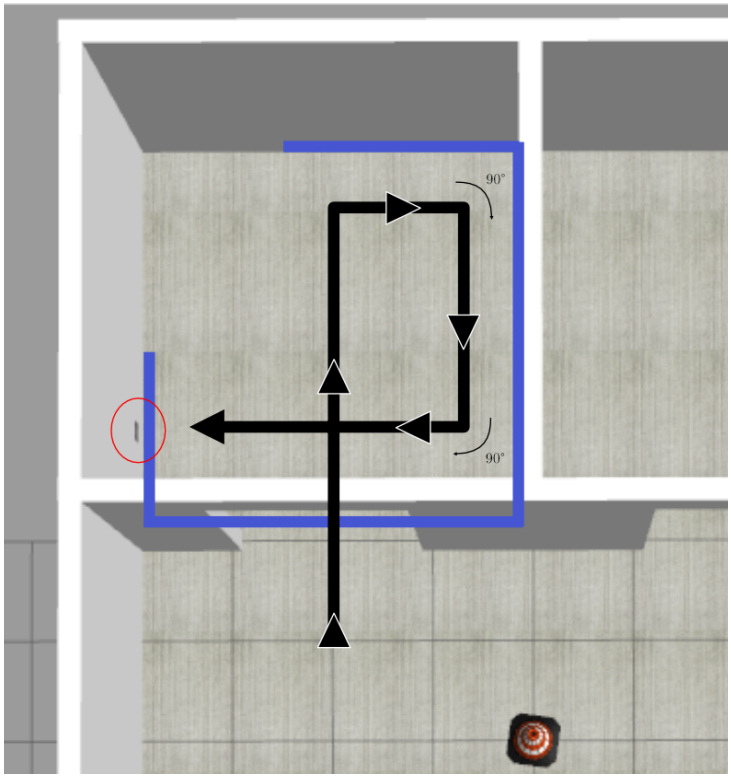


Figure 6 — Exploration Routine

Map Saver

When the DogBot returns to the starting point it has to save the map obtained from the mapping node. To avoid manual operations to do so, the planner node publishes on the "\salvamappa" topic a string, which is then read by the Map Saver node. Its role is to write a .png file of the explored environment and the info of the image in a .yaml file. A modified version of the Map Saver node of the Map Server ROS package has been implemented to achieve the described behaviour.

In figure the png generated after two experimental cases. The first map generated after exploring the sequence room1 - room3 - room2 before finding the desired marker and returning to the starting position. The latter generated by exploring room 1 - room 3 before finding the QR Code.

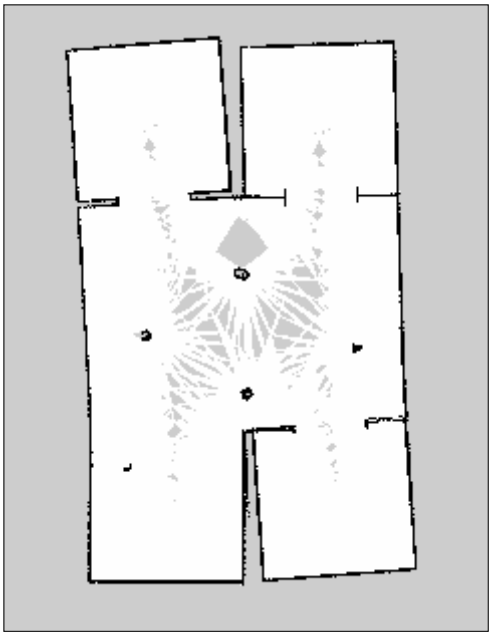


Figure 7.1 — Environment Map

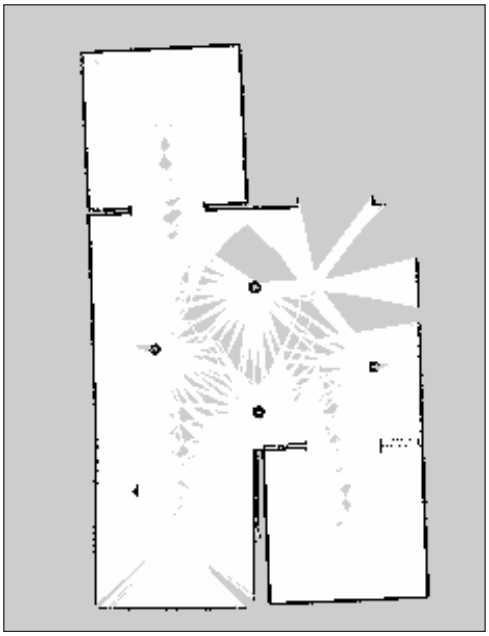


Figure 7.2 — Environment Map

QR Code Detection

QR Codes have been generated using the github repo cpp_qr_to_png [8], which combines two projects, one to generate QR Codes, the other to write a png file in c++. The repo has been copied into the project to easily create and modify the QR Codes.

OpenCV node task is to detect the QR Marker and retrieve its encoded ID. The QR Code detection node relies on OpenCV and ZBar library. To ensure that the marker is detected there's a two steps process, first the camera data is elaborated via OpenCV functions, then ZBar is used to decode the data encrypted in the marker.

The image sensor data is passed to OpenCV via CvBridge needed to convert between ROS Image messages to OpenCv images. Is then converted to GreyScale and then using the threshold function binarised. The first choice for the threshold is a fixed value set to 50, so all the pixels of the matrix with values above it are converted into white pixels. Considered that the lighting is uniform a fixed threshold value is sufficient, another viable solutions is to implement adaptive thresholding and filtering the image for noise. To dynamically set the threshold Otsu's Method is then been implemented. Both the last approach and the fixed threshold method gave satisfying results, the second one is then been chosen for the final version of the code from which the filtered images are shown.

The binarised image is then scanned for symbols by the ZBar scan function. When the text encrypted in the QR Marker is detected is then published on a topic read by a subscriber running on the global planner node. To retrieve camera image data this node subscribes to "/camera/ImageRaw", while it publishes the decrypted QR Code as a string on "opencv/marker_data".

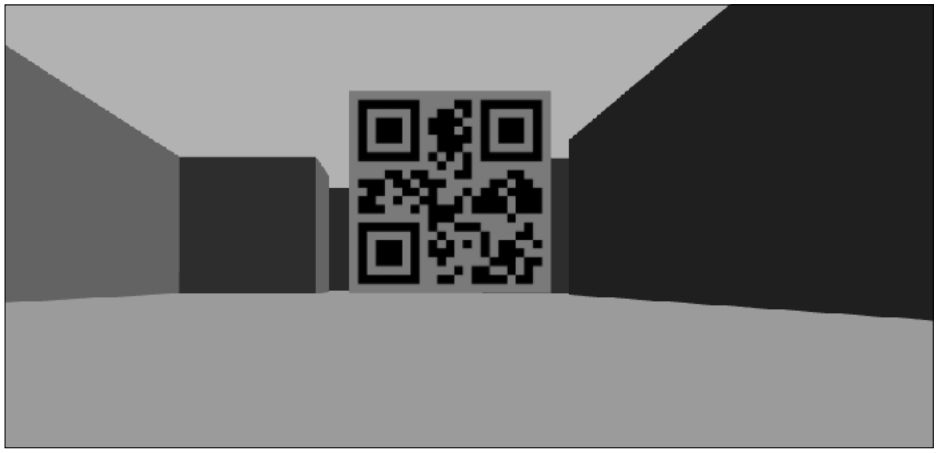


Figure 8.1 — QR Marker filtered

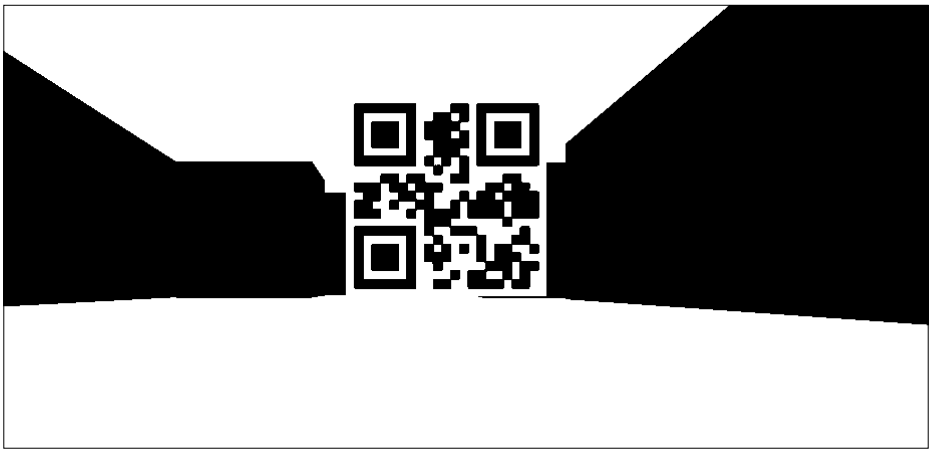


Figure 8.2 — QR Marker filtered

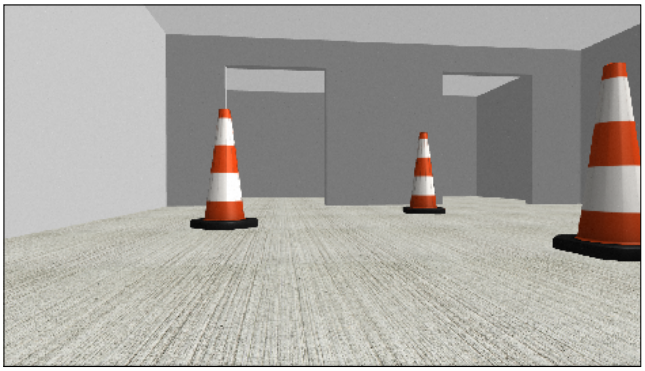


Figure 9.1 — Navigation Camera Unfiltered

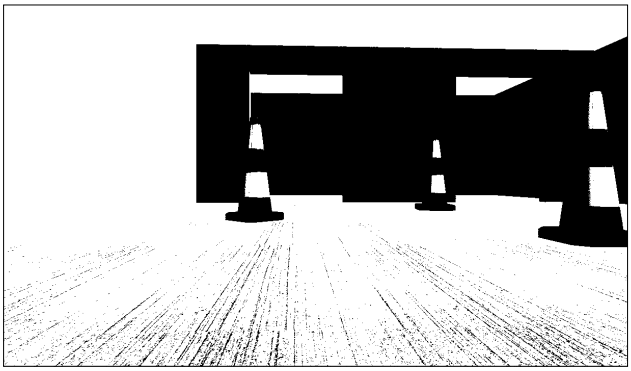


Figure 9.2 — Navigation Camera Filtered

Force Plugin

To test DogBot's behaviour when subjected to external forces a gazebo model plugin is developed and implemented. The force plugin runs automatically when the robot's xacro is loaded and exerts a constant force of 1N along the x-axis of the world. To achieve this a class "DogbotExtForce" is created by inheriting ModelPlugin. It is then instantiated a pointer to robot's base_link to retrieve its pose and apply the force to it. In order to run it we then register it with GZ_REGISTER_MODEL_PLUGIN().

Results

Study Cases

The system setup was tested with various case scenarios to verify robustness of both the control and navigation systems, with multiple cases for obstacle avoidance and local minimum exceptions. Study cases included different room orders, different values for the starting position, different configurations of the obstacles in the open space. Also QR Marker positions and height were modified to test the effectiveness of the exploration routine. Camera resolution have also been lowered to 480p to test both the exploration routine and the QR Code detection.

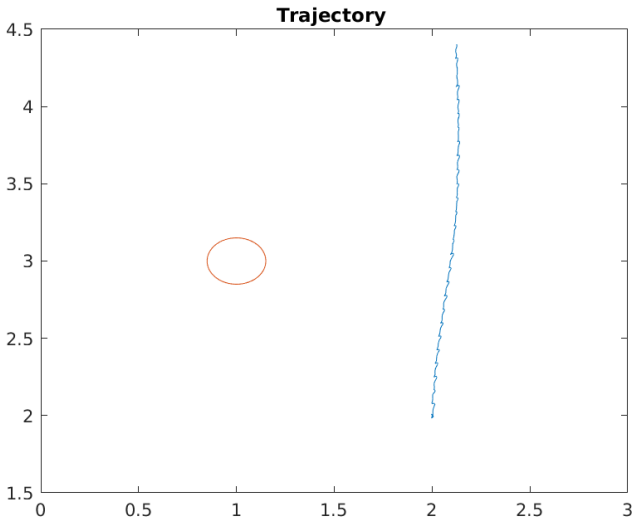
Simulations were run in two cases, with both the force plugin turned on and explicitly passing the wrong value of the robot's total mass underestimating it by 10%, and in ideal conditions without external forces and with the correct value for the mass.

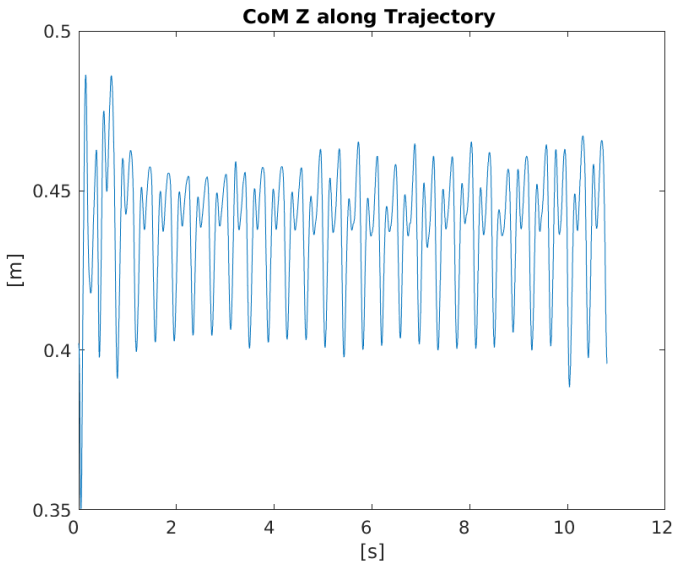
Data from the simulation are extracted by publishing the values on the topic "dogbot/debug" using a custom message, all useful topics are then recorded during the simulation as rosbag files.

All the recorded data is imported in Matlab, elaborated and finally plotted through a script.

Graphs

The following graphs show the results obtained with the discussed package. Plot refers to a segment of the trajectory followed during the video simulation. Two simulation cases are considered: a first set of graphs shows the ideal conditions simulation, the second group shows a simulation with an external force and mass underestimation.

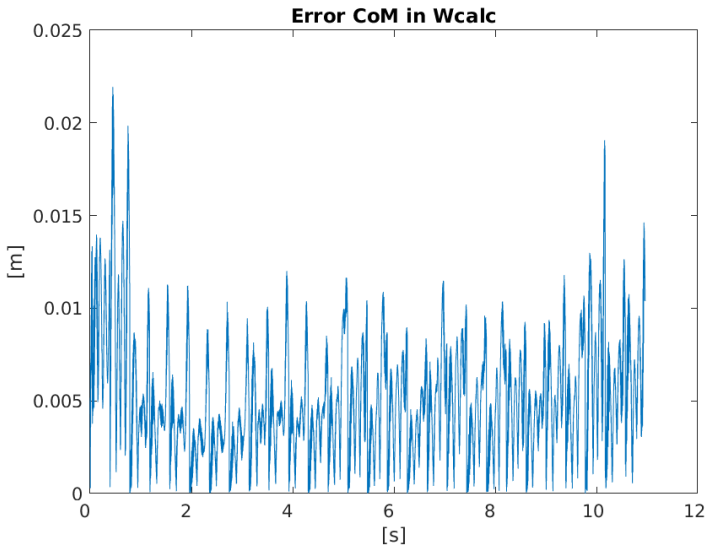


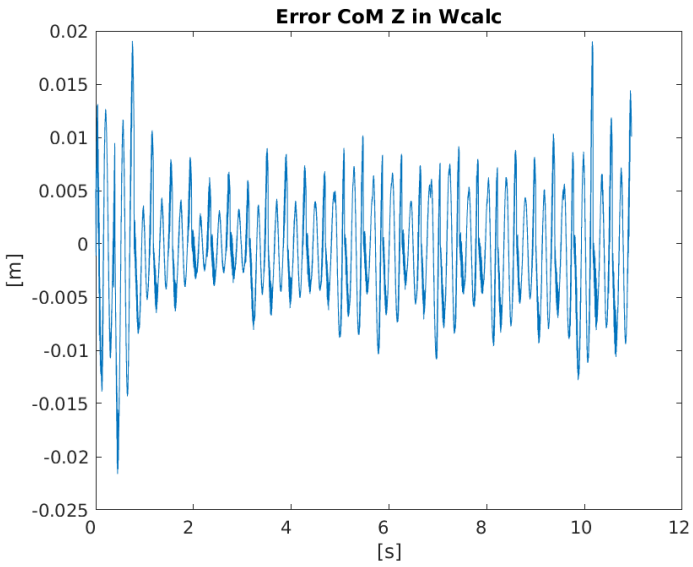


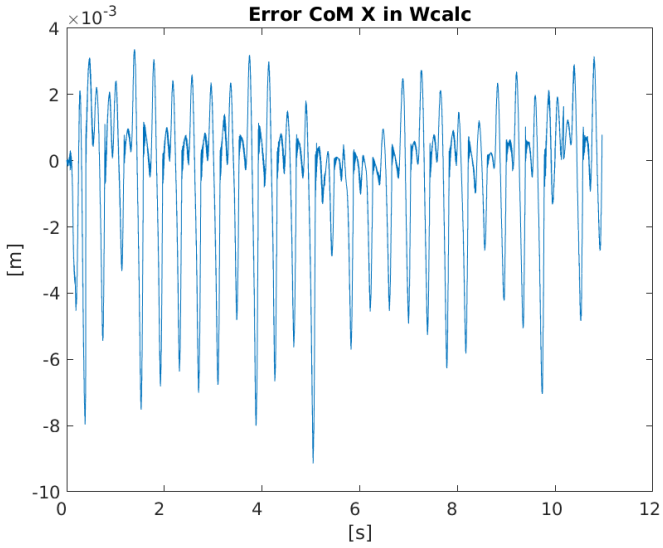
As shown robot starts at (2,2) and the computed repulsive force — for the presence of the construction cone in (1,3) — pushes it on the right. The first value for the z-component of the CoM is outside the normal range values as there's a delay between the start of the gazebo simulation and the first iteration of the control loop. This case shows that the stack control+towr is also robust to variations of the initial condition.

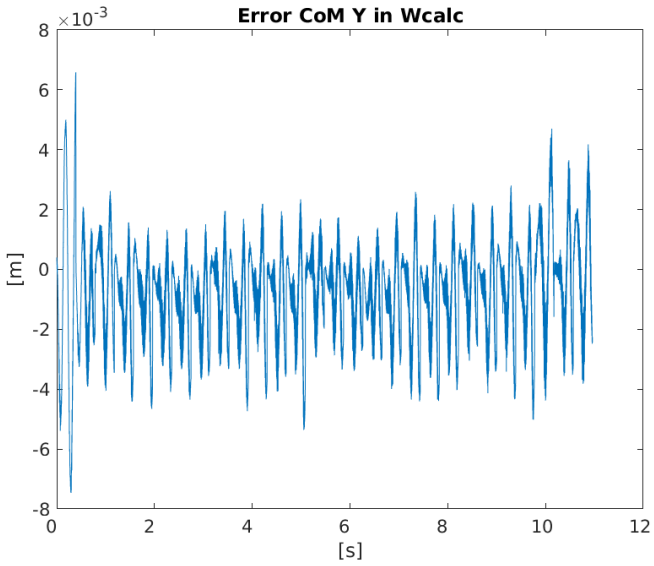
Rewriting the equation for the computation of the desired wrench on the CoM, the following graphs show the error dynamics for the positional term $(r_{c,ref} - r_c)$ and velocity term $(\dot{r}_{c,ref} - \dot{r}_c)$.

$$w_{com,des} = K_p(r_{c,ref} - r_c) + K_d(\dot{r}_{c,ref} - \dot{r}_c) + mg + M_{com}(q)\ddot{r}_{c,ref}$$

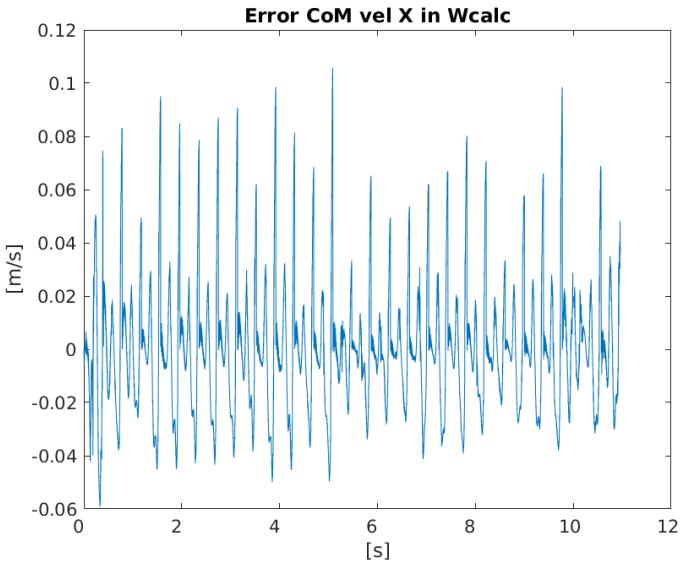


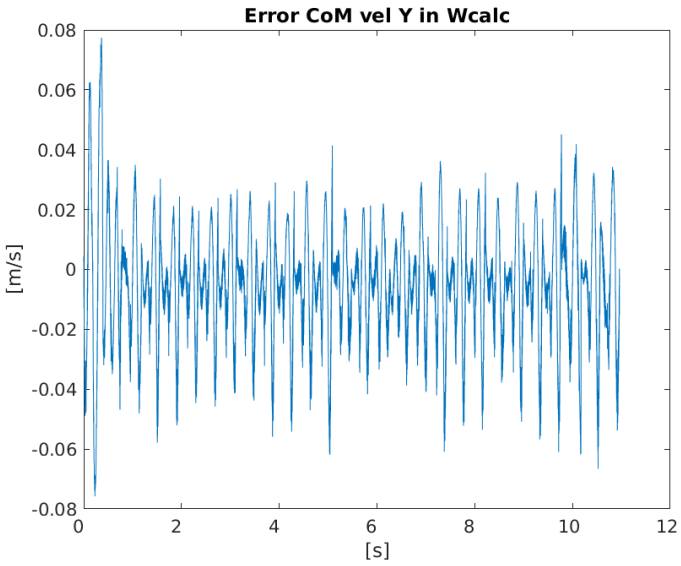


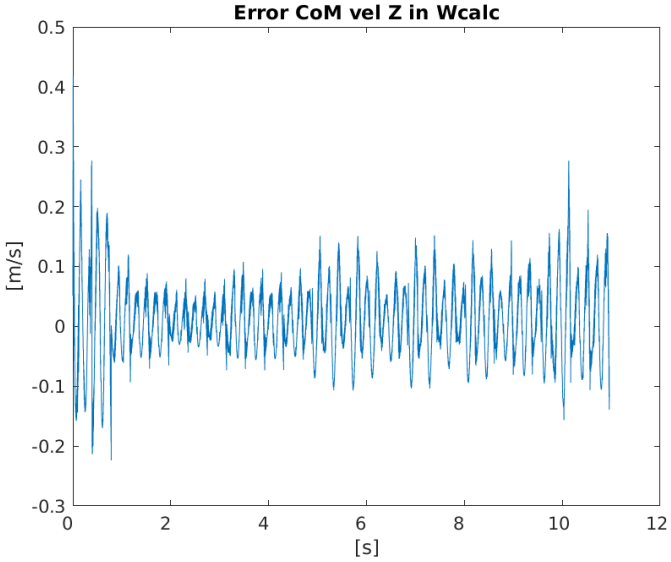


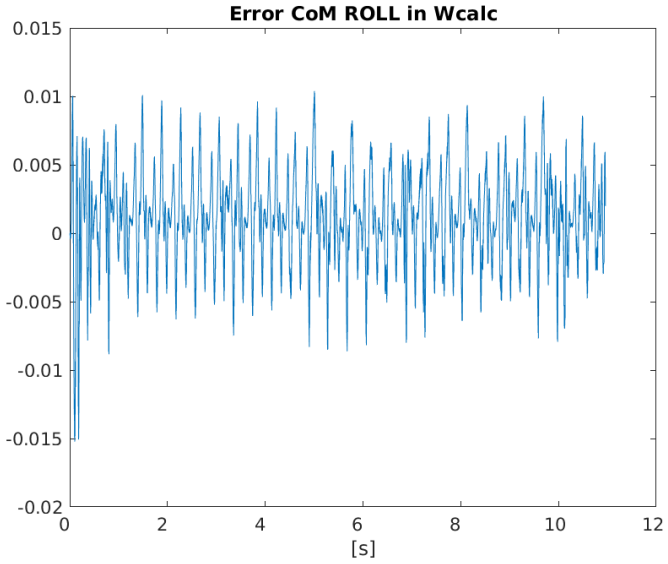


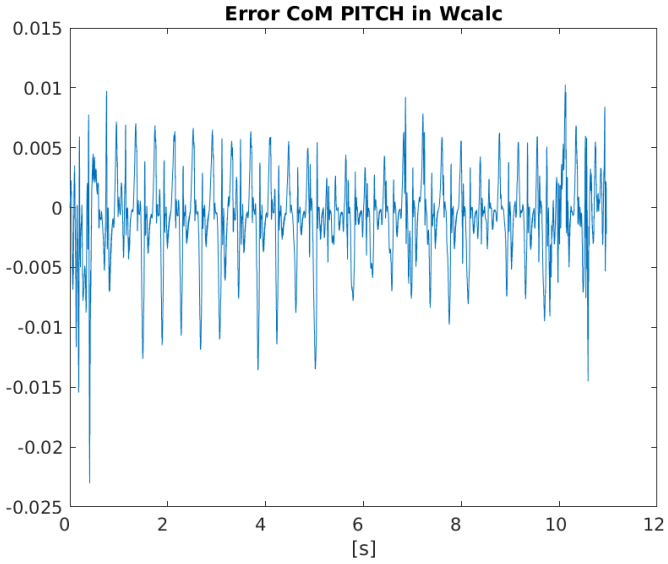
Whilst x and y references are tracked with an error of the 10^{-3} order, the z error order is 10^{-2}, resulting in a total error for the CoM position of the same order. This is explained as the reference for x and y components change slower than the z reference. Analogous considerations apply for the following velocity terms.

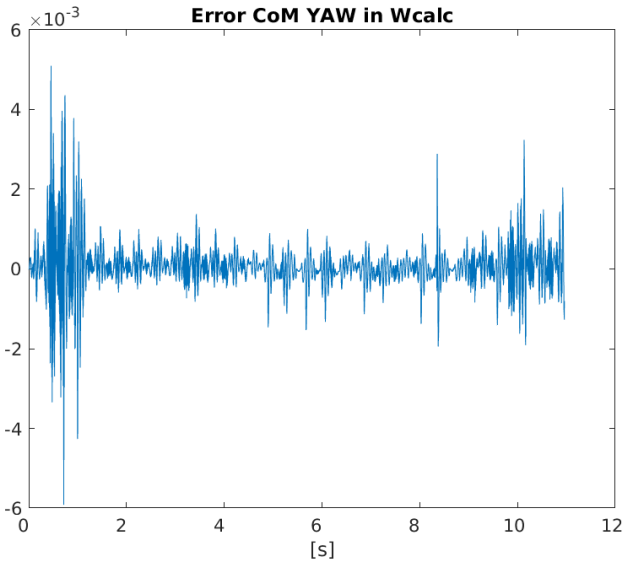

Error CoM vel X in Wcalc
[m/s]
[s]
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
0
2
4
6
8
10
12


Error CoM vel Y in Wcalc
[m/s]
[s]
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
0
2
4
6
8
10
12

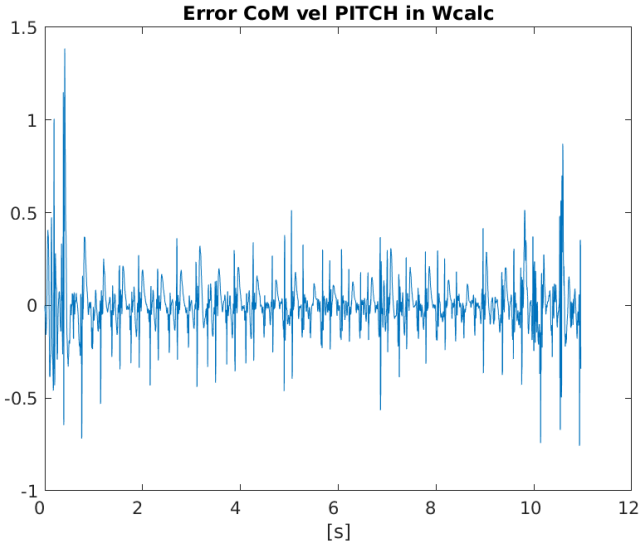

Error CoM vel Z in Wcalc
[m/s]
[s]
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
0
2
4
6
8
10
12

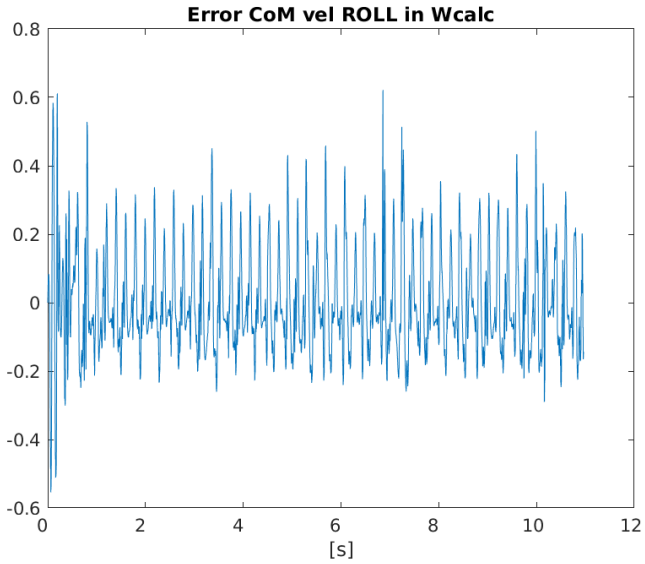


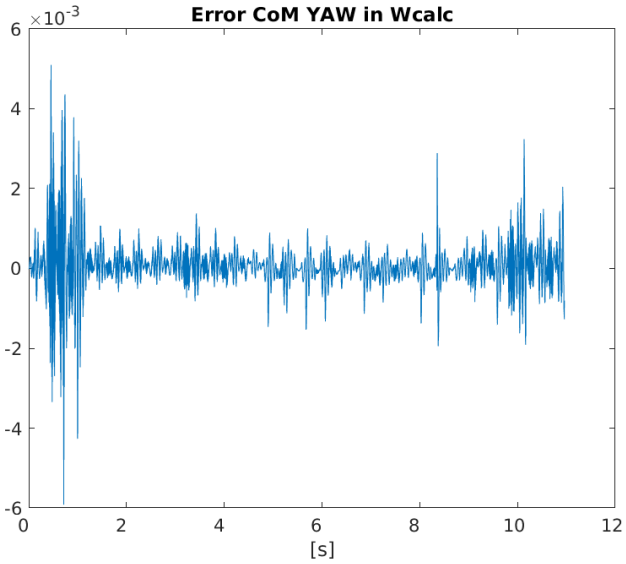




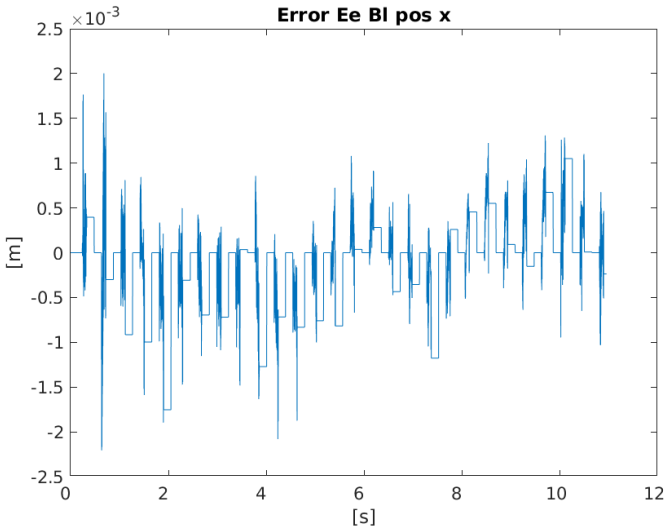
Orientation Error for the heading direction is always less than 1/3 of a degree, where pitch and roll errors are slightly greater. Analogous considerations apply for the velocities.

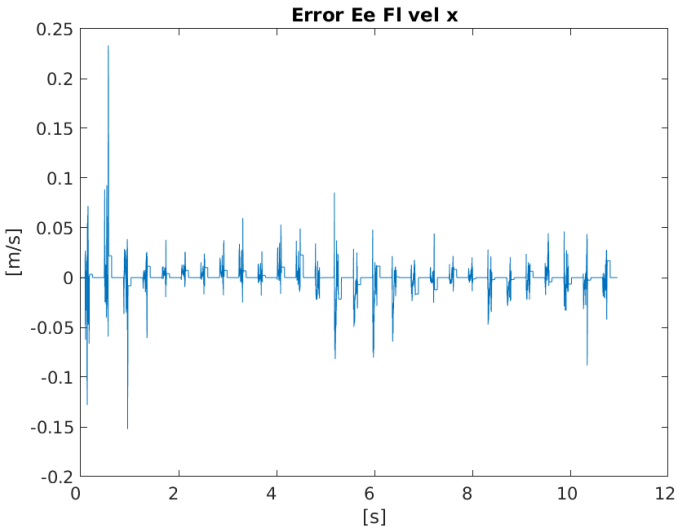


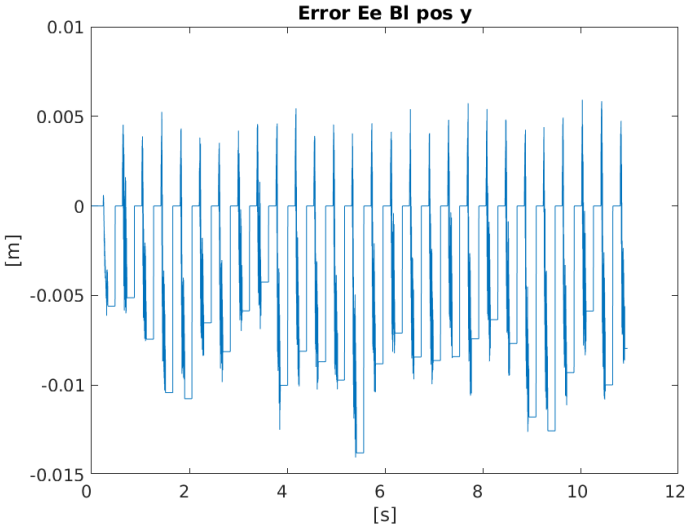


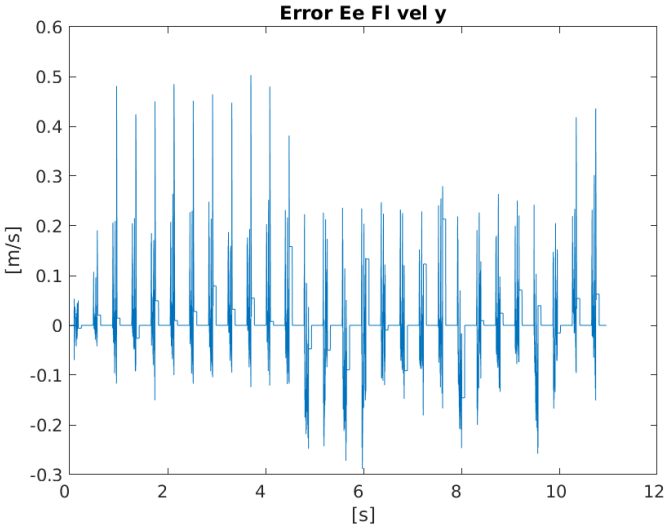


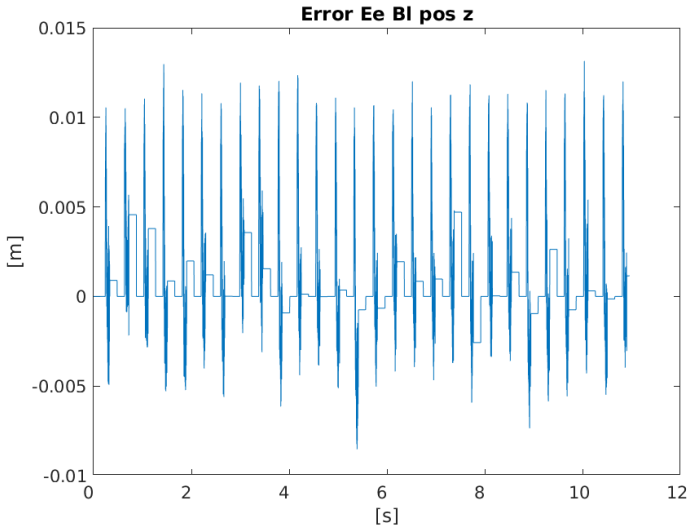
The error for the pose of a single foot is shown in the following graphs. As the robot is moving along the y axis, the error along the x component is lower. As in 1 second the gait trot moves twice the same foot, there are 2 spikes which are the result of the kick off impulse to move the feet from the ground, followed by a period of zero error. Anyway the error is in the range of the centimetre or millimetre.

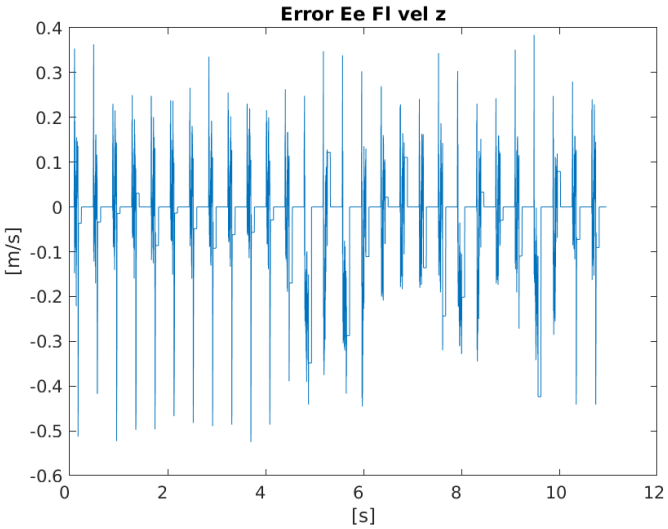

Error Ee Bl pos x
×10^-3
[m]
[s]


Error Ee Fl vel x
[m/s]
[s]

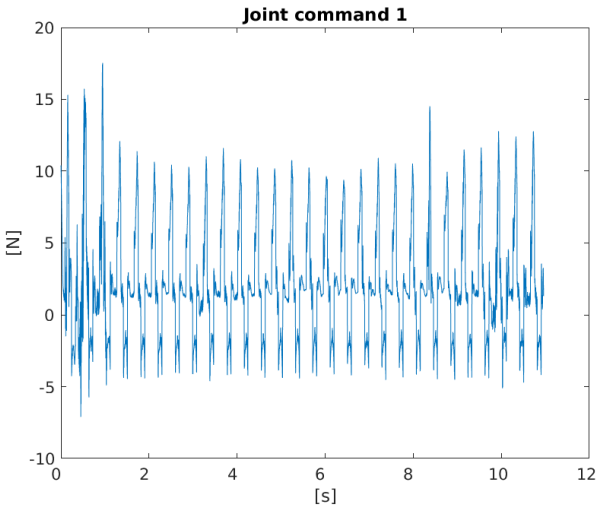

Error Ee Bl pos y
[m]
[s]

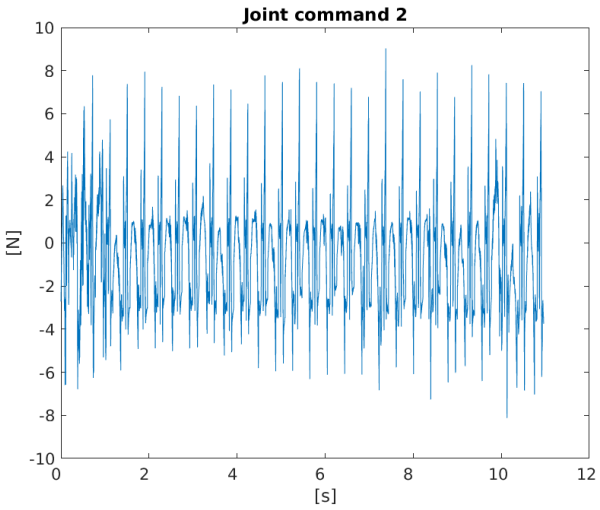

Error Ee Fl vel y
[m/s]
[s]

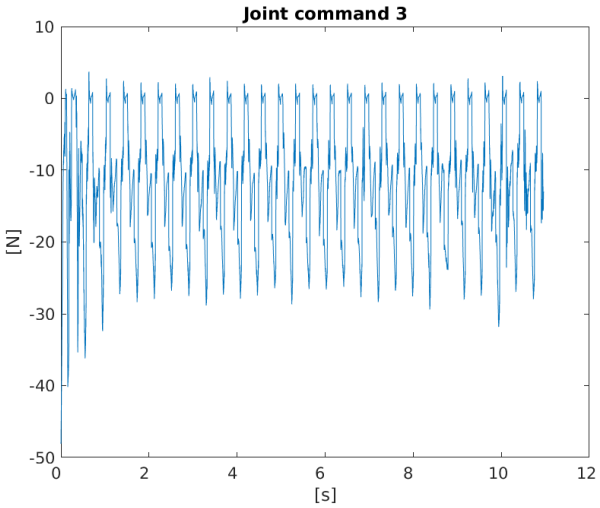

Error Ee Bl pos z
[m]
[s]

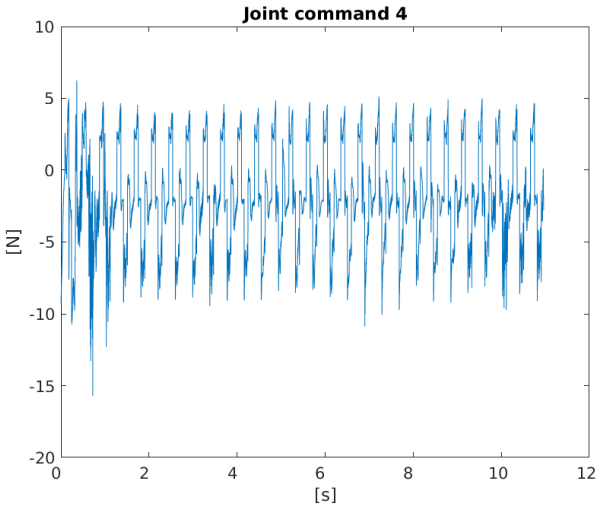

Error Ee Fl vel z
[m/s]
[s]

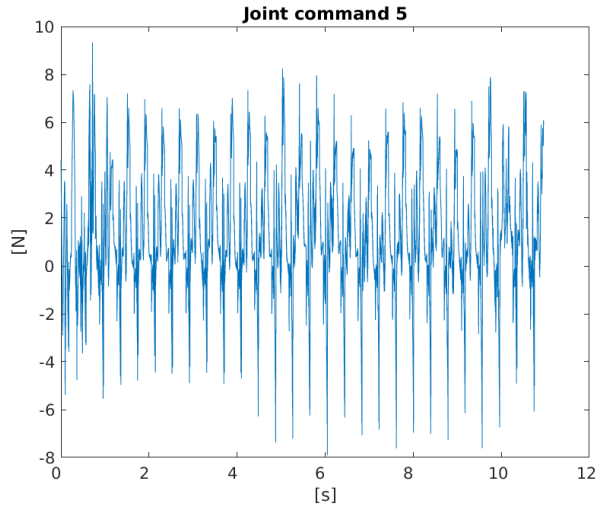
Graphs show joints torques commands never exceed the saturation value of 60N.

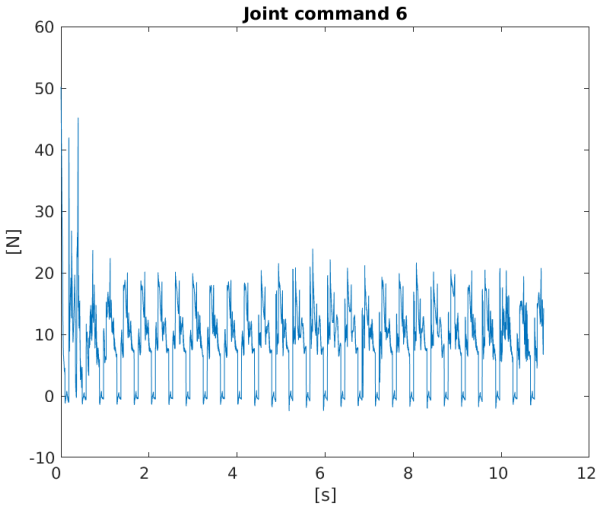


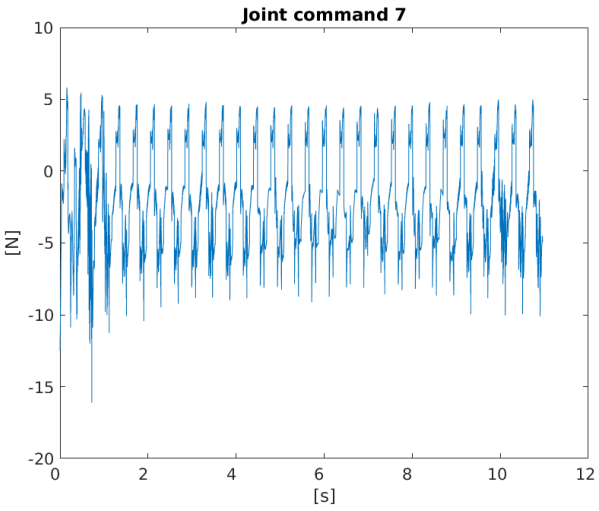


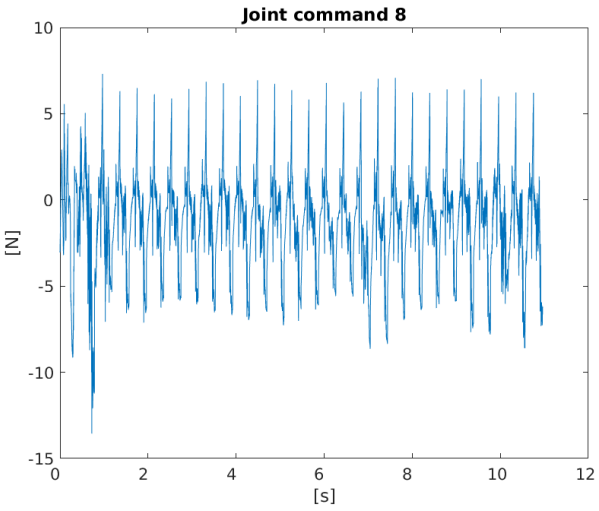


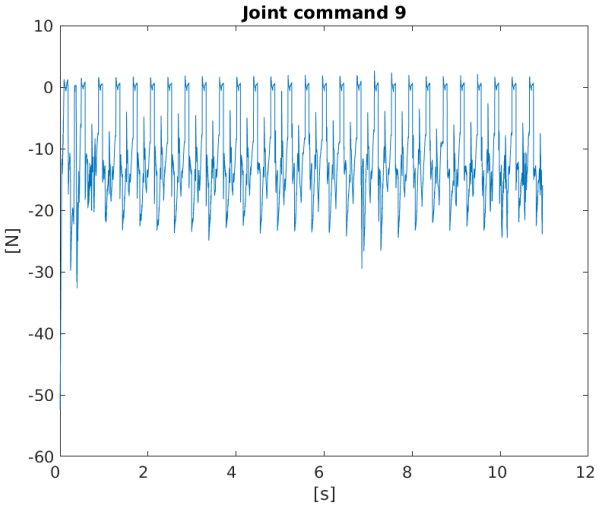


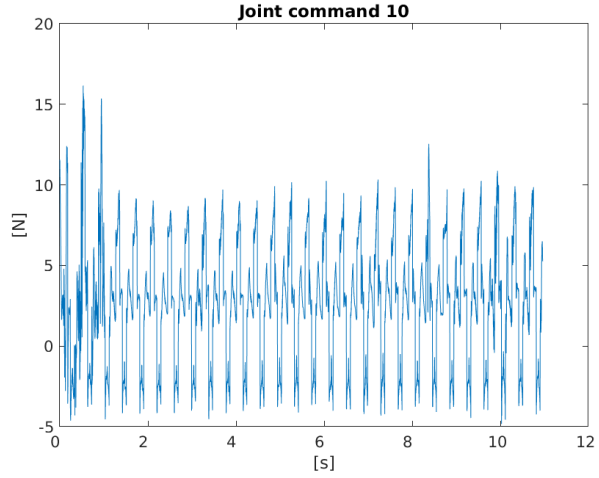


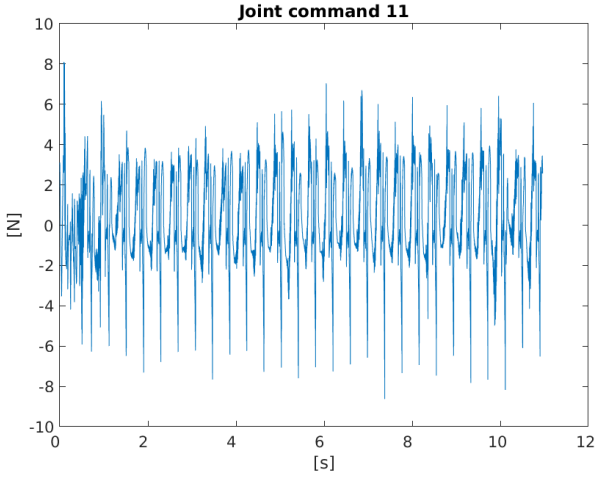


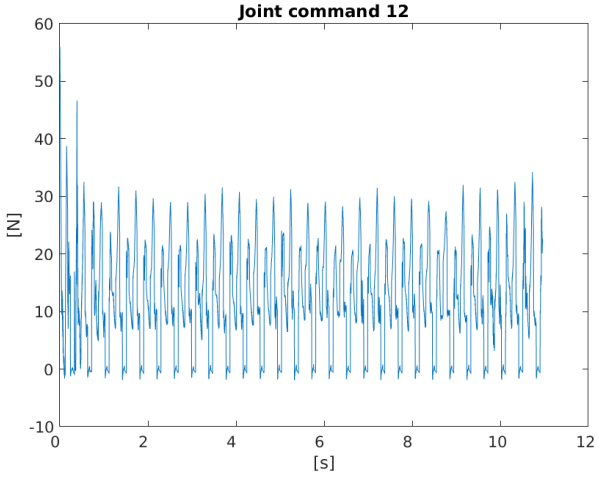

Joint command 7
[N]
[s]

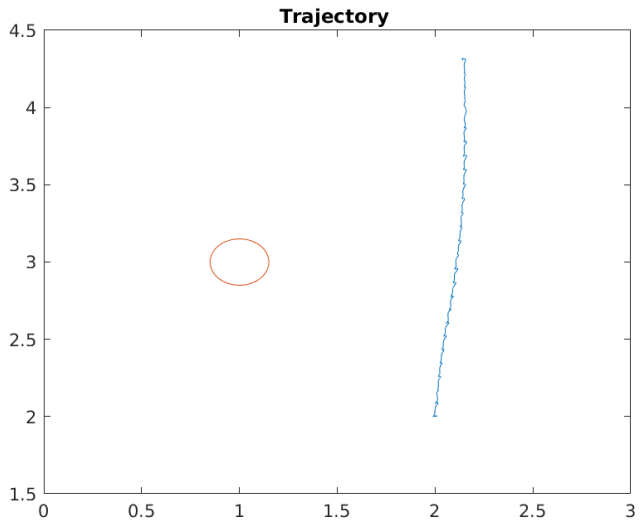

Joint command 8
[N]
[s]

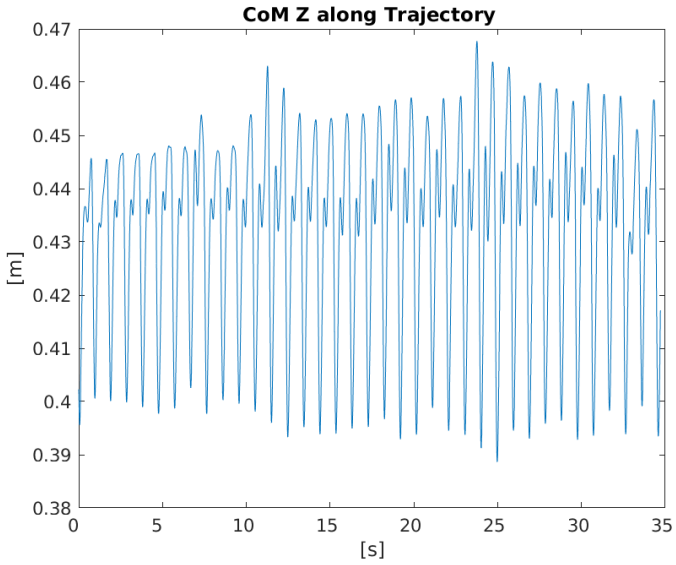

Joint command 9
[N]
[s]

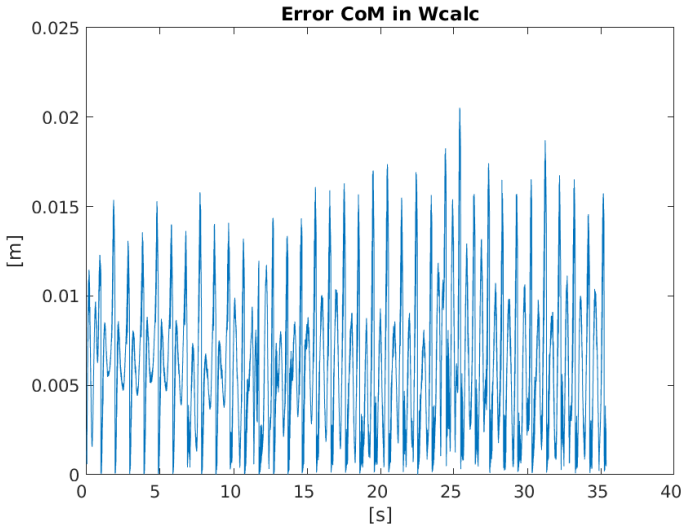

Joint command 10
[N]
[s]

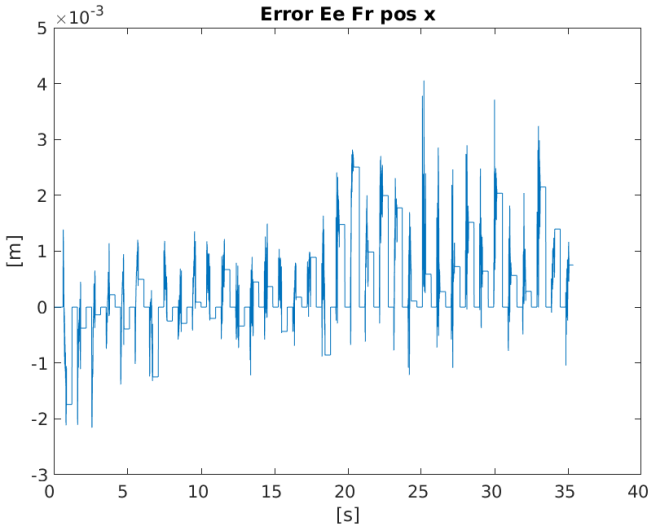

Joint command 11
[N]
[s]

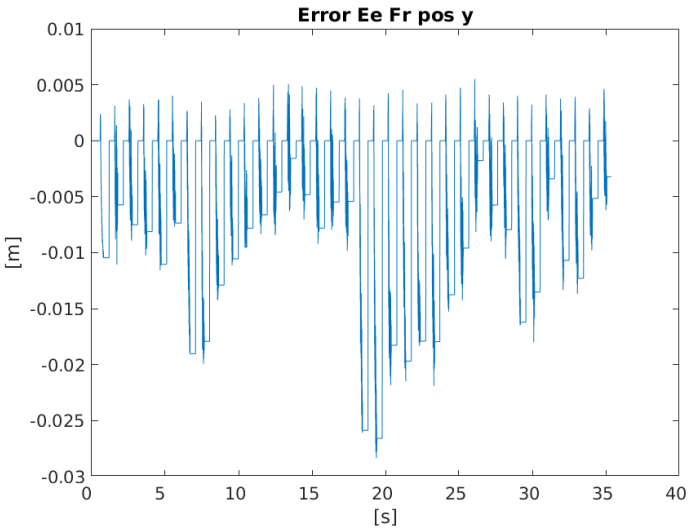

Joint command 12
[N]
[s]

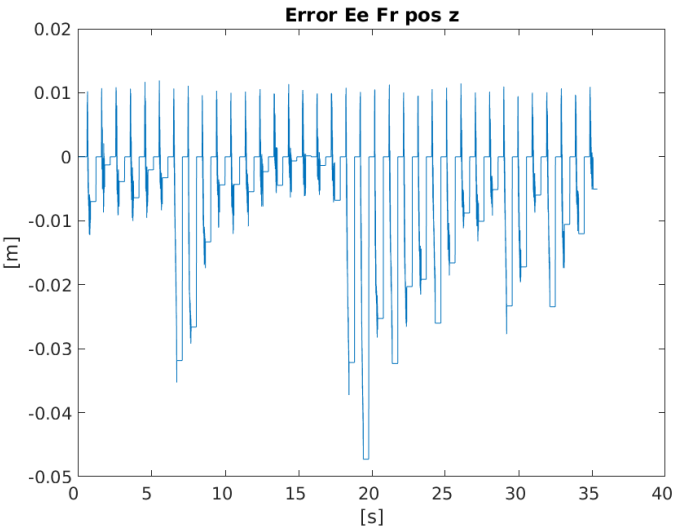

Trajectory

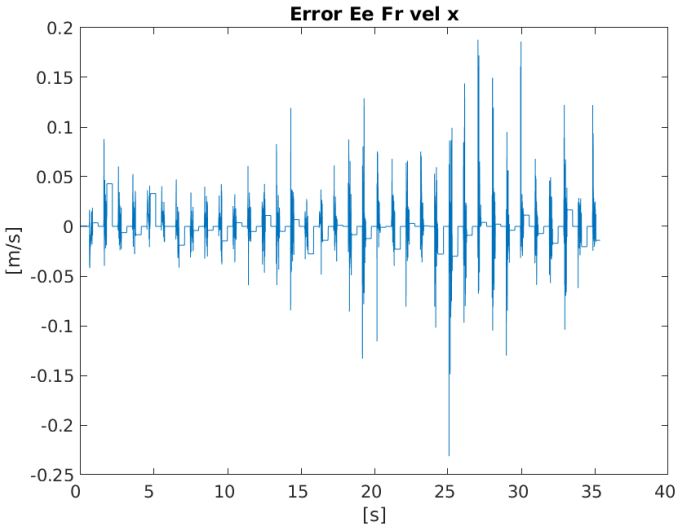

CoM Z along Trajectory
[m]
[s]

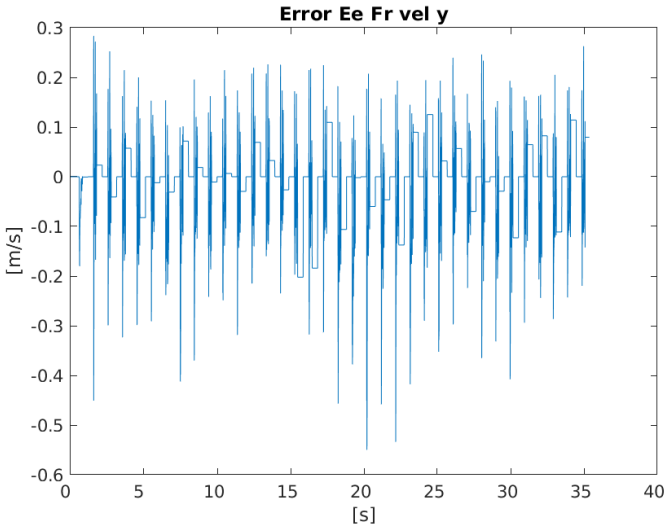

Error CoM in Wcalc
[m]
[s]

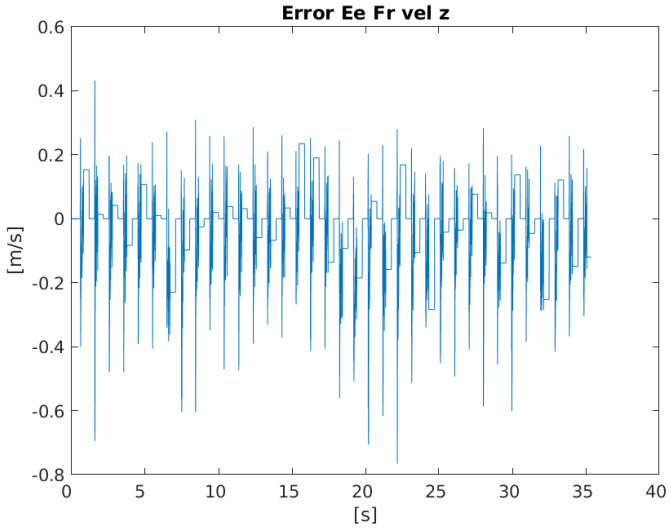

Error Ee Fr pos x
×10^-3
[m]
[s]

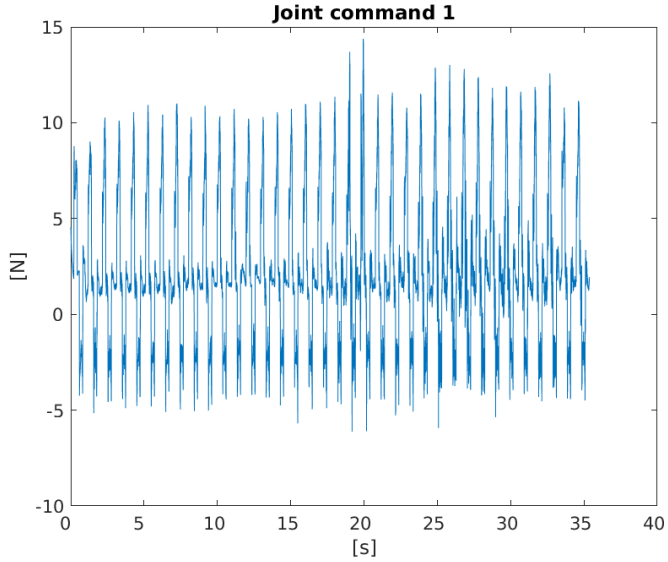

Error Ee Fr pos y
[m]
[s]

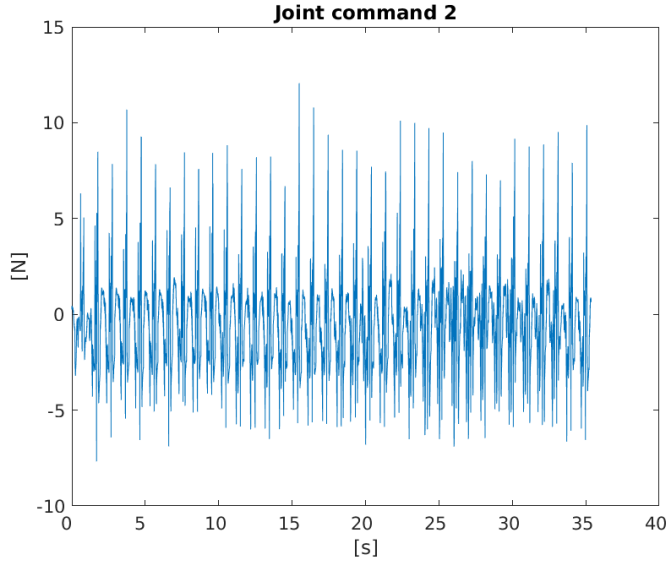

Error Ee Fr pos z
[m]
[s]

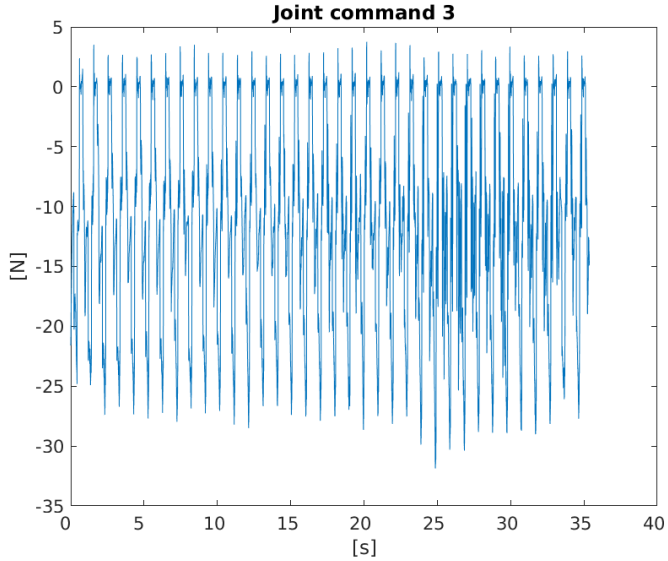

Error Ee Fr vel x
[m/s]
[s]


Error Ee Fr vel y
[m/s]
[s]

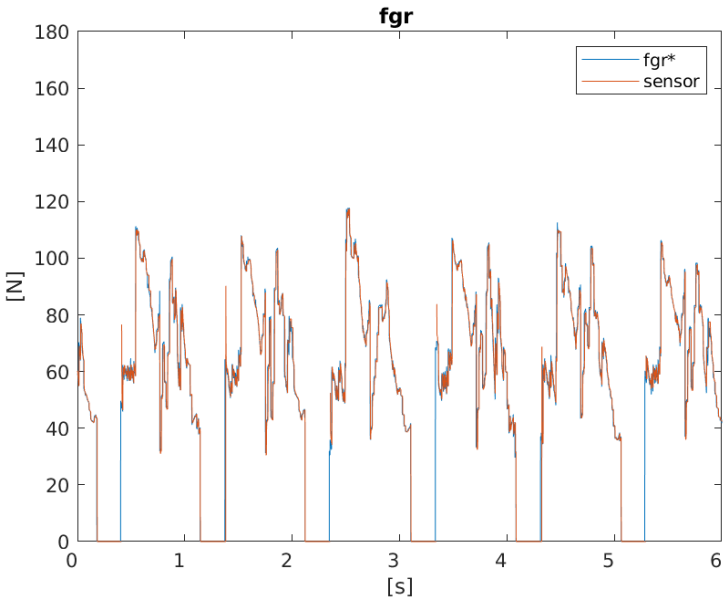

Error Ee Fr vel z
[m/s]
[s]

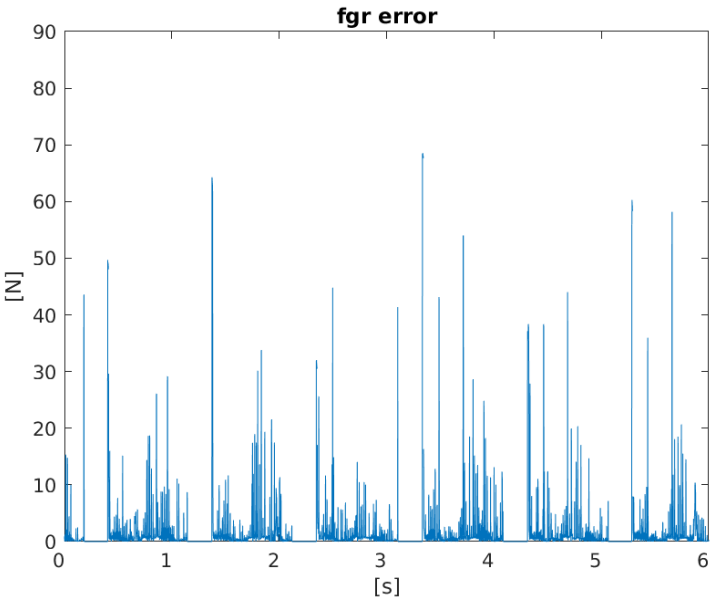

Joint command 1
[N]
[s]

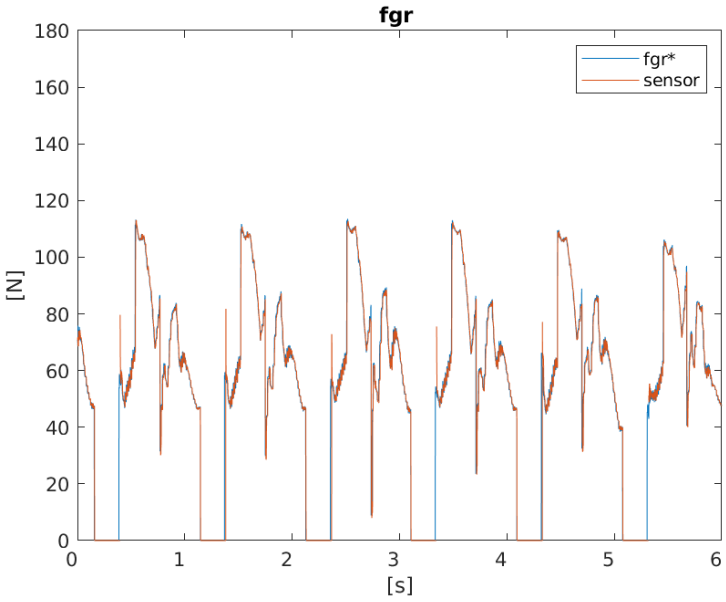

Joint command 2
[N]
[s]

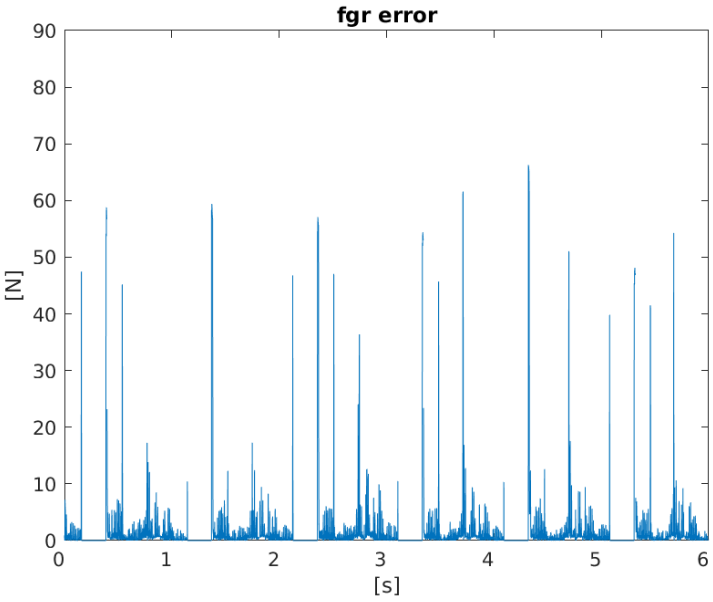

Joint command 3
[N]
[s]

In the following plots are shown the ground reaction force $f_{gr}^{\star}$ computed with the controller, and the actual force detected by the contact sensor. All plots are for the hind left foot. The same two cases are shown, the first two graphs refer to the simulation with the external forces and the underestimated mass value, while the graphs after refer to the ideal condition case.









Webography

[1] https://github.com/ReactRobotics/DogBotV4
[2] V. Morlando, A. Teimoorzadeh, F. Ruggiero, "Whole-body control with disturbance rejection through a momentum-based observer for quadruped robots", Mechanism and Machine Theory, vol. 164, 104412, 2021, DOI: 10.1016/j.mechmachtheory.2021.104412.
[3] https://www.alglib.net/optimization/quadraticprogramming.php
[4] http://docs.ros.org/en/kinetic/api/towr/html/index.html
[5] https://github.com/ethz-adrl/ifopt
[6] https://coin-or.github.io/Ipopt/
[7] https://github.com/robotology/idyntree
[8] https://github.com/RaymiiOrg/cpp-qr-to-png