UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

Project for a Field and Service Robotics Course

Motion Planning and Control of a Differential-drive Robot

Professor:
Fabio Ruggiero

Candidate:
Andrea Cavaliere
Matr. M58/240

Anno Accademico 2019/2020

Abstract

The following project shows the results achieved by the author controlling a differential-drive robot using a probabilistic algorithm and a graph search technique, followed by the implementation of two different controllers. All the project has been implemented in ROS melodic framework and simulation runs on Gazebo 9.0(and RViz for demonstration purpose).

Contents

List of Figures

Chapter 1

Introduction

1.1 Robot: Turtlebot 3 Burger

TurtleBot is a differential-drive robot, based on ROS platform with a very well documented libraries and tools available. In the very first stage of this project a different robot had been chosen but, due to the lack of detailed information and appropriately modeled .xacro files has been replaced by Tb3. The model used in this project is "Burger" but any model is suitable with the right correction of kinematic and dynamic parameters.

1.1.1 Dimension

Turtlebot3 Burger is a very small differential-drive robot.

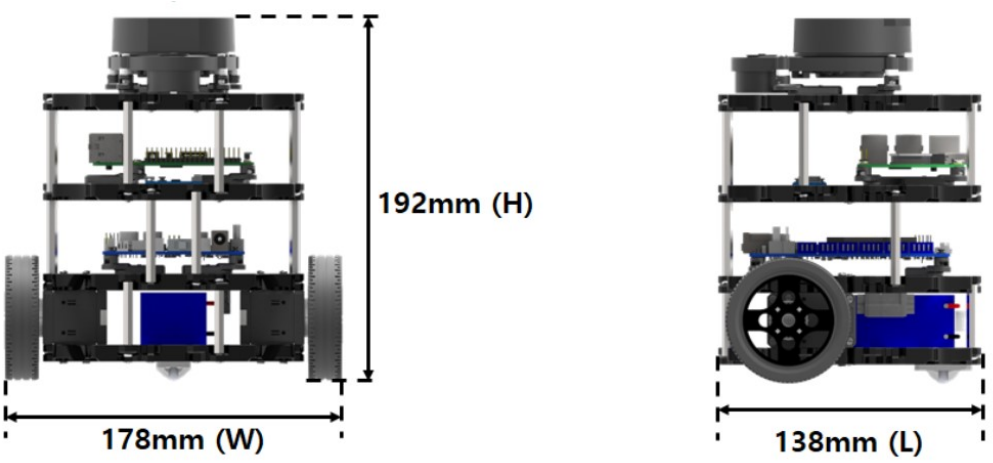


Figure 1.1: Turtlebot3 Burger Main Dimensions.

Seen in a 2-D Cartesian Space the robot has a circular shape with 2 front-wheel at a distance of 16 cm from each other which implies, considering the rotation oaround the vertical axis, a circular space with a diameter of $21cm$ (see Fid. 1.2)

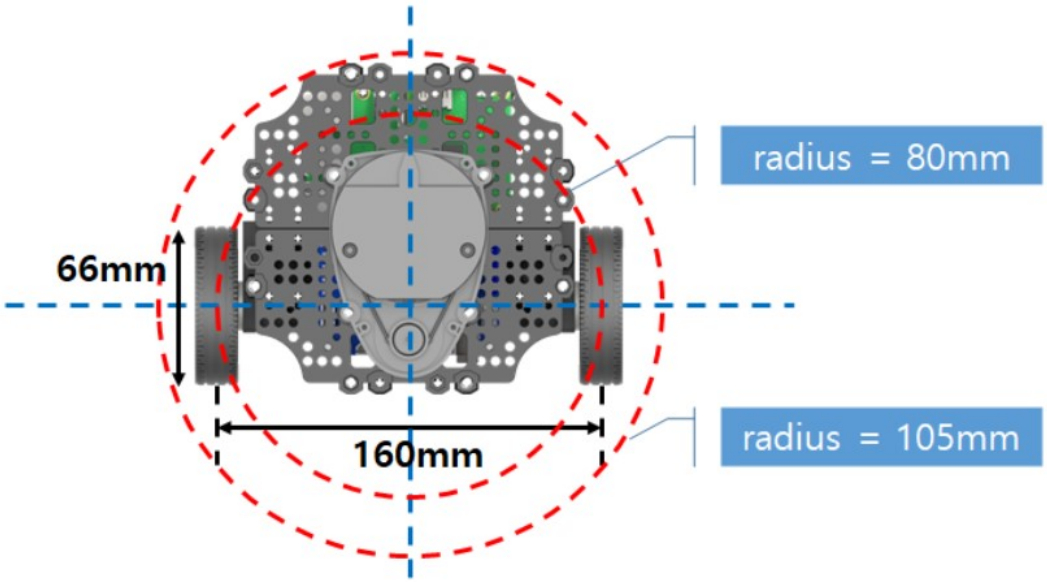


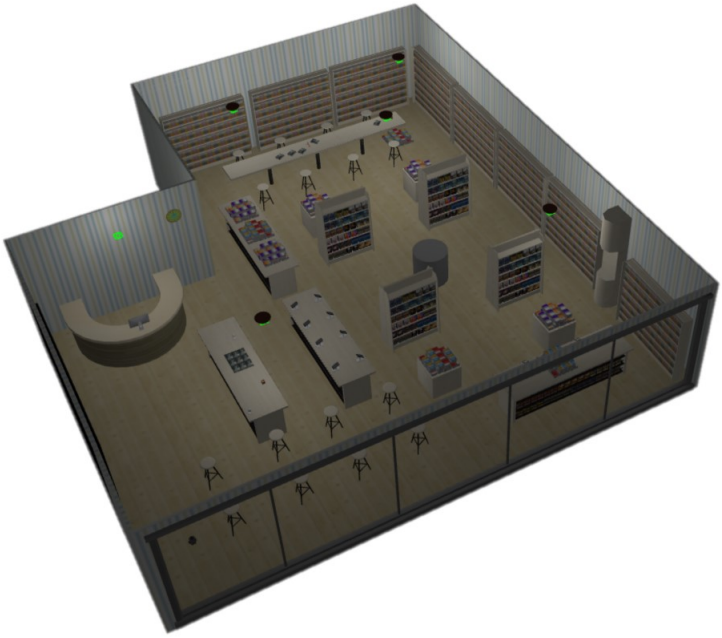
Figure 1.2: Turtlebot3 Burger from above.

1.1.2 Hardware Specifications

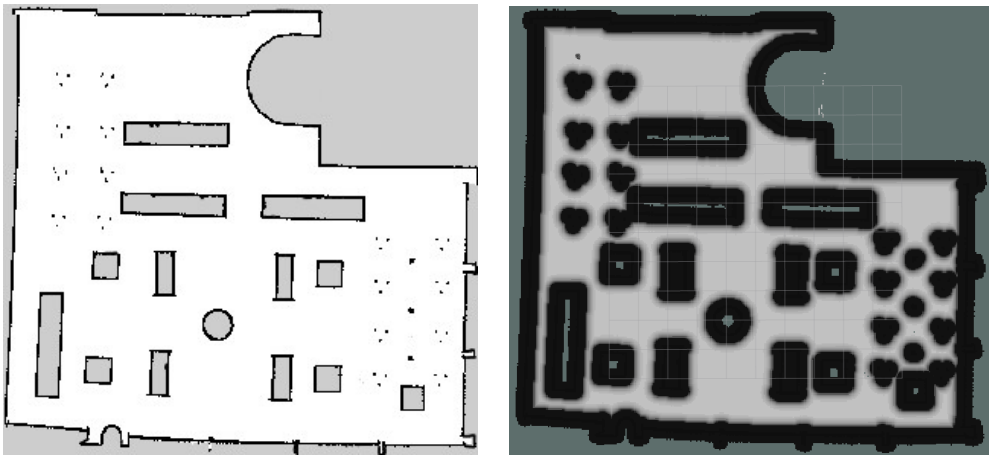
Weighting only $1Kg$, the turtlebot3 Burger has a maximum linear velocity of $0.22m/s$ and a maximum rotational velocity $2.84rad/s$. [5] The robot is equipped with a 2-D Laser Distance Sensor LDS-01 with $360degree$ range, thanks to this sensor it has been possible to build the map in the first stage of the project.

1.1.3 Environment

Thanks to Amazon Web Service Robotics[1], it was possible to test the robot in a realistic environment. The robot explore a simulated bookstore with many obstacles. Since the project also required the use of circular obstacles, the environment was partially modified by removing two obstacles and adding the required one. The environment is about $100m^2$ large centred in the origin of world frame.



Before being able to test both planner and controllers developed, the map of the environment is required.
To do so, the robot has been driven around each part of the room building the map thanks to the SLAM algorithm [3]. After that, each obstacle undergoes an isotropic growth equal to a slightly higher value than the robot radius. The results of this procedure are shown in Fig.1.3.



(a) Map of the environment (b) resulting costmap

Figure 1.3: map visualization before and after isotropic growth

Chapter 2

Motion Planning

In the following, the author will first introduce a brief description of the robot model examinated and then the two steps needed to create a path that allows the latter to reach a destination starting from an initial condition.

2.1 Robot Model

Since it's a differential-drive robot the kinematic model is equivalent to the unicycle model:

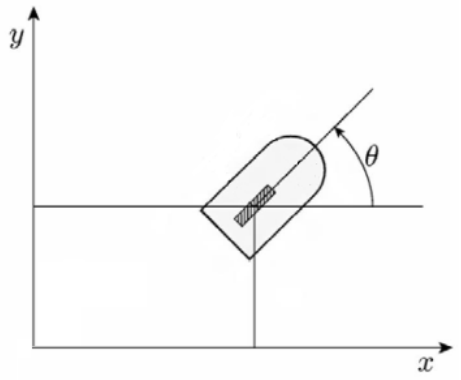


Figure 2.1: Unicycle model

Let's define the state-space vector as $[x\, y\, \theta]^T$ where the first two components are the Cartesian coordinates of the contact point of the wheel with the ground and the last one is the orientation of the wheel with respect to the x axis.[4] In order to make the two systems equivalent it must be considered that in a differential robot the vector of the inputs is defined by $[\omega_r\, \omega_l]$, respectively right and left . in a unicycle instead it is $[v\, \omega]$. For this reason the following transformation is necessary:

$$v = \frac{r(\omega_R + \omega_L)}{2} \quad \omega = \frac{r(\omega_R - \omega_L)}{d} \tag{2.1}$$

the following model is therefore obtained:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) \\ sin(\theta) \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{2.2}$$

2.2 Bidirectional RRT

Bidirectional RRT is a single-query probabilistic method which makes use of a data structure called RRT (Rapidly-exploring Random Tree). In addition to RRT expansion procedure from the *source node* the algorithm builds a second tree, simultaneously with the first, rooted in the *goal node* and then a second phase called *connection procedure* begins.

$BuildTree(q_{init})$;
$T = initializeTree(q_{init})$;
for $i = 1$ **to** $MaxIteration$ **do**
  $q_{rand} = randomState()$;
  $Extend(T, q_{rand})$;
end
return T
$Extend(T, q)$;
$q_{near} = nearestNeighbor(q, T)$;
if $newConfig(q, q_{near}, q_{new})$ **then**
  $T.addVertex(q_{new})$;
  $T.addLink(q_{near}, q_{new})$;
  if $q_{new} = q$ **then**
    return $REACHED$
  else
    return $ADVANCED$
  end
end
return $TRAPPED$
$Connect(T_{source}, T_{goal})$;
while $S \neq REACHED$ **do**
  if $PreviousTree = T_{source}$ **then**
    $q_{rand} = randomState()$;
    $Extend(T_{source}, q_{rand})$;
    $S = Extend(T_{goal}, q_{source,new})$;
  else
    $q_{rand} = randomState()$;
    $Extend(T_{goal}, q_{rand})$;
    $S = Extend(T_{source}, q_{goal,new})$;
  end
end

In the pseudocode written above is shown all the steps of a Bidirectional

RRT algorithm. The idea behind this code is very simple and effective. First, the two trees are built with a fixed number of nodes and then begin a procedure where, while one of them keep its expansion, the other one tries to connect to the new node using it as its own q_{rand} in the expansion procedure. Usually the algorithm ends when a connection between the trees is found. This work very well in particular environment where one, or at least a very few number of *gates* connect q_{source} and q_{goal}. In this particular case a single connection not always ensure a good solution but this will be better discussed in the last chapter.
Typical results are showed in the videos in the github folder. [2]

2.3 A* Algorithm

A* was developed4 during the *Shakey project*. It is a *best-first* search algorithm aiming to find a path between a starting and a goal node in a graph with the lowest cost, i.e. minimum time or least distance travelled or safest path in particular cases. Being more specifical, A* build the path that minimizes:

$$f(n) = g(n) + h(n) \tag{2.3}$$

where $g(n)$ it the cost of path from source node to node n and $h(n)$ is called *heuristic* function, i.e a function which estimates the cost of the path from node n to goal node,in this way it can evaluates a straight line distance or maximize distance from obstacles. In this project $h(n)$ estimates distance from n to goal since $\mathcal{C}_{free}$ has been carefully chosen. Below it's possible to read the pseudocode:

```
A*Search(T, q_goal)
while OPEN.isnotempty() do
    extract(q_best) ;
    if q_best = q_goal then
        return REACHED
    else
        for i = 0 to i = #links of q_best do
            if q_i is unvisited then
                OPEN.add(q_i);
                q_i.parent = q_best;
            else
                if g(q_best) + distance(q_best, q_i) < g(q_i) then
                    q_i.parent = q_best ;
                end
                if q_i ∉ OPEN then
                    OPEN.add(q_i);
                else
                    updatef(q_i);
                end
            end
        end
    end
end
```

In this project Bidirectional-RRT will build a roadmap where each node is connected to others with straight lines ans so will be the resulting path. This choice reduces computation time of algorithm, simplifies structure of the tree and, since is a very cluttered environment, ensure a safe connection following a linear path.

Chapter 3

Motion Control

In the following chapter two type of controllers will be shown, first a brief introduction about each one and relative advantages and drawbacks.

3.1 Posture Regulation

As title suggests, this kind of controller drives the robot to a desired configuration q_d. In this kind of situation the turtlebot solves a regulation problem from a node to the next one in the generated path. Posture regulation is able to controll the whole configuration of the robot (Cartesian position and orientation).

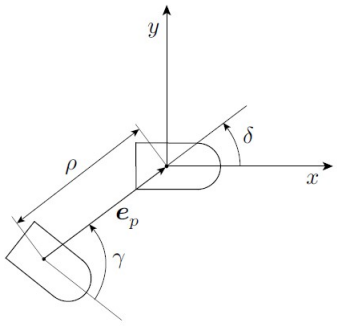


Figure 3.1: Polar rappresentation of Posture Regulation.

in 3.1 the main parameters of the problem are shown. Vector e_p represents the Cartesian error between actual position and desired one, γ denotes the orientation error between actual configuration and e_p, similarly δ indicates the orientation error between e_p and desired orientation of the robot. Assuming that the the desired configuration is the origin $q_d = [0\,0\,0]^T$ the value of ρ δ and γ can be derived:

$$\rho = \sqrt{x^2 + y^2}$$
$$\gamma = Atan2(y, x) - \theta + \pi$$
$$\delta = \gamma + \theta$$

Based on this representation, the kinematic model of the unicycle is expressed as:

$$\dot{\rho} = -v cos(\gamma)$$
$$\dot{\gamma} = v\frac{sin(\gamma)}{\rho} - \omega$$
$$\dot{\delta} = v\frac{sin(\gamma)}{\rho}$$

Finally let's define the control law as follows

$$v = k_1 \rho cos(\gamma)$$
$$\omega = k_2\gamma + \frac{cos(\gamma)sin(\gamma)}{\gamma}(\gamma + k_3\delta)$$

It is worth noting that such controllers can assume undefined value when $\gamma = 0$ but with a very simple and well-known mathematical property

$$\frac{cos(\gamma)sin(\gamma)}{\gamma} = \frac{sin(2\gamma)}{2\gamma} \longrightarrow 1 \quad for \quad \gamma \longrightarrow 0 \tag{3.1}$$

Starting from our velocity constraints introduced in 1.1.2 is possible to notice a remarkable difference in the values of v and ω, so with the right values of k_1, k_2 and k_3 it's possible to generate a particular behaviour of the robot where it seems to rapidly reorient itself on the spot and then increase the linear velocity to reach destination. From a maximum value of $\rho = 0.8$ the values of gain are the following $k_1 = 0.2\, k_2 = 1.2$ and $k_3 = 0.8$ setting the desired yaw of each intermediate node in the path $\theta_{des,i} = Atan2(e_{p,y}, e_{p,x})$

3.2 Input/Output Feedback Linearization

Differently from previous controller, a trajectory tracking controller will now be introduced.
A common and simple choice is based on Input/Output Feedback Linearization. Since differential-drive robots are kinematically equivalent to unicycle ones it's possible to consider the following outputs:

$$y_1 = x + bcos\theta$$
$$y_2 = y + bsin\theta$$
$$b \neq 0$$

Where $|b|$ represents the distance of a point B placed along the sagittal axis from the contact point of the wheel.

$$\begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} cos(\theta) & -bsin(\theta) \\ sin(\theta) & bcos(\theta) \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = \mathbf{T}(\theta) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.2}$$

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} cos(\theta) & -bsin(\theta) \\ -\frac{sin(\theta)}{b} & \frac{cos(\theta)}{b} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \mathbf{T}^{-1}(\theta) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{3.3}$$

Equations 3.2 and 3.3 will help in the last phase of the control to translate the virtual inputs $[u_1\, u_2]$ into physical ones $[v\, \omega]$. from these equations it is possible to derive the following representation:

$$\dot{y_1} = u_1$$
$$\dot{y_2} = u_2$$
$$\dot{\theta} = \frac{u_2 cos\theta - u_1 sin\theta}{b}$$

From which it's possible to obtain the control law:

$$u_1 = \dot{y_{1,d}} + k_1(y_{1,d} - y_1)$$
$$u_2 = \dot{y_{2,d}} + k_2(y_{2,d} - y_2)$$
$$k_1 > 0,\, k_2 > 0$$

From these expressions it's clearly visible that whole evolution of θ is not controlled by virtual inputs $[u_1\, u_2]$ since no orientation error in computed. For this reason, since project requires a posture control, this *I/O FBL* will be supported by a posture regulation controller.

Chapter 4

Results

In the last chapter the author will introduce results achieved with both controllers and compare them trying to highlight advantages and drawbacks of each one. Both controllers runs at $100Hz$ update frequency, so does the planner. About the bidirectional-RRT it starts creating two trees with 50 nodes each with $0.8m$ long edge between each node and then begin *connection procedure*.
In this second phase the algorithm follows a so called *greedy* behaviour with a limitation on the result edge length.Normally this kind of behaviour has no limitation and only stops if q_{new} or an obstacle is reached but in this case, since there is a high number of obstacles and can't ensure safety of the robot from one trajectory segment to the next one.

4.1 Simulation

4.1.1 Posture Regulation

Since the path is made of linear connection between few important nodes and project aim to control orientation only in the last one, the simplest controller realization is *Posture Regulation* where every intermediate nodes have orientation computed starting from orientation of vector joining each vertex with the previous one, directed towards the next one. in this way, as it can be clearly deduced by Fig. 4.3, Turtlebot rapidly reorients itself almost on the spot and then travel to the next destination.

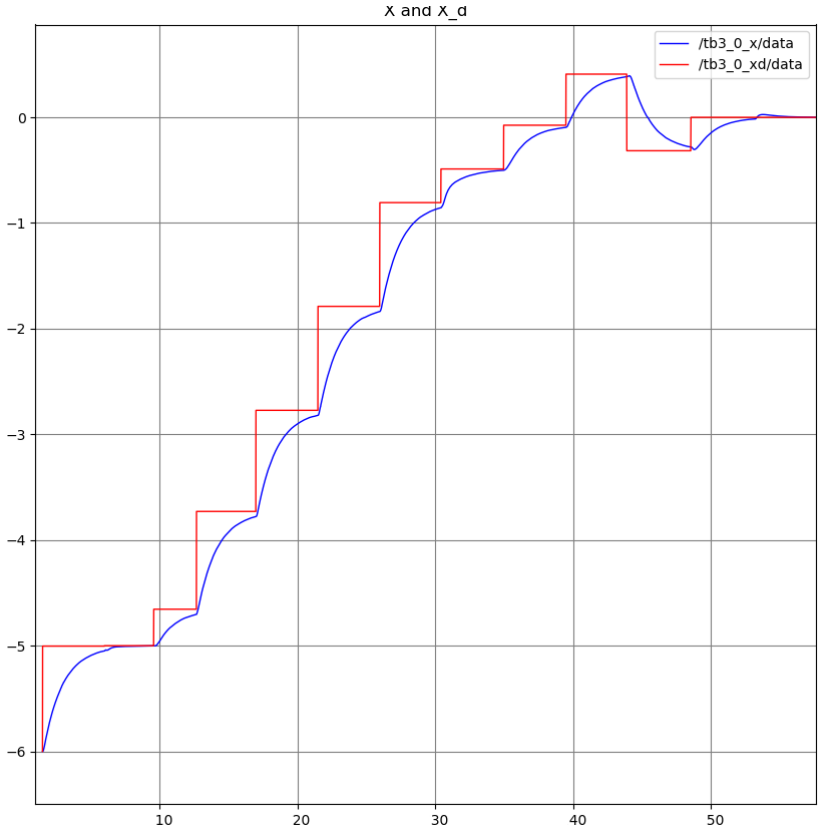


Figure 4.1: Evolution of coordinate x and x_d

Main advantages are that linear velocity v is always positive semi-definite and robot strictly follows planned trajectory. Since one of the necessary conditions is $\rho \neq 0 \forall t$, Cartesian error can decrease to a fixed resolutions but will never reach 0.
On the other hand, as in every trajectory tracking problem, a regulation controller doesn't ensure the best performance, in Fig. 4.3 and Fig. 4.4 there are conspicuous and repetitive peaks every time it reach a node, moreover velocity is linearly dependent from ρ so it goes to zero as long as ρ decreases so it could lead to a significant loss of time.

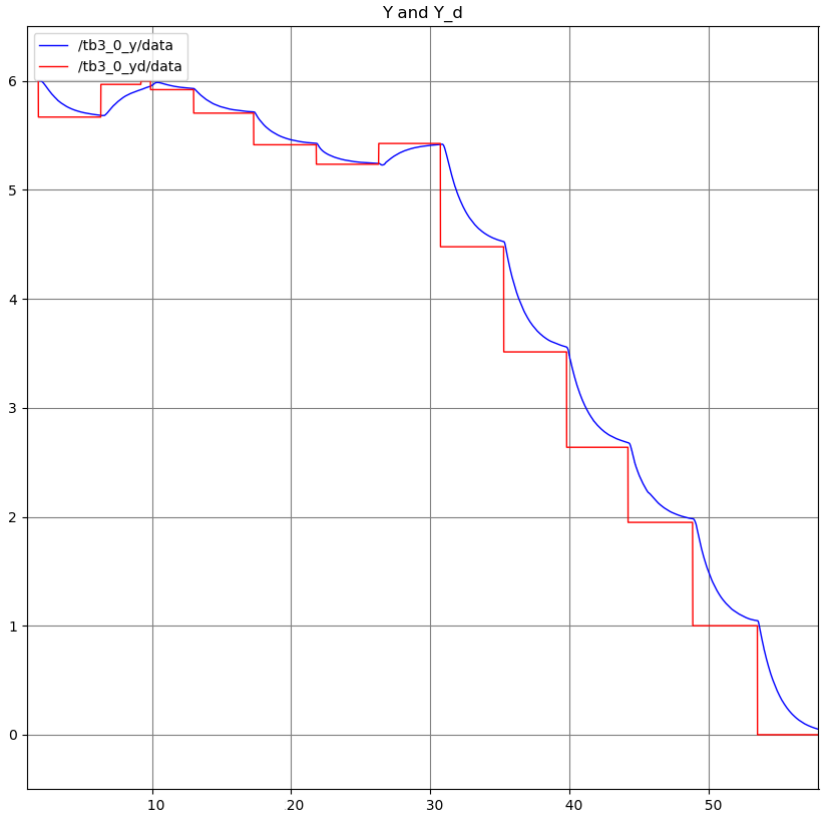


Figure 4.2: Evolution of coordinate y and y_d

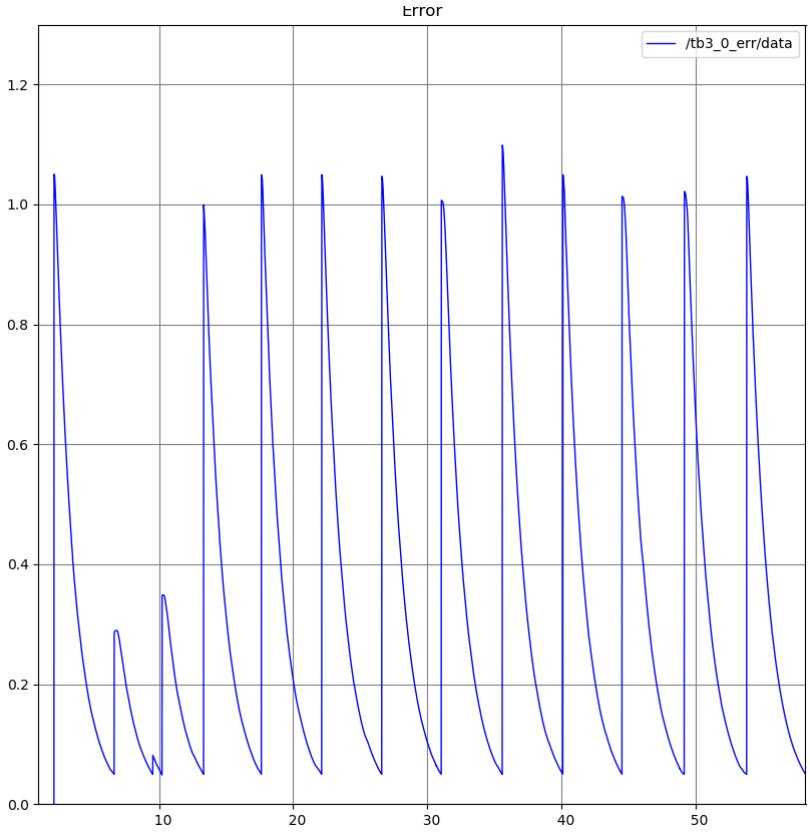


Figure 4.3: evolution of Cartesian error

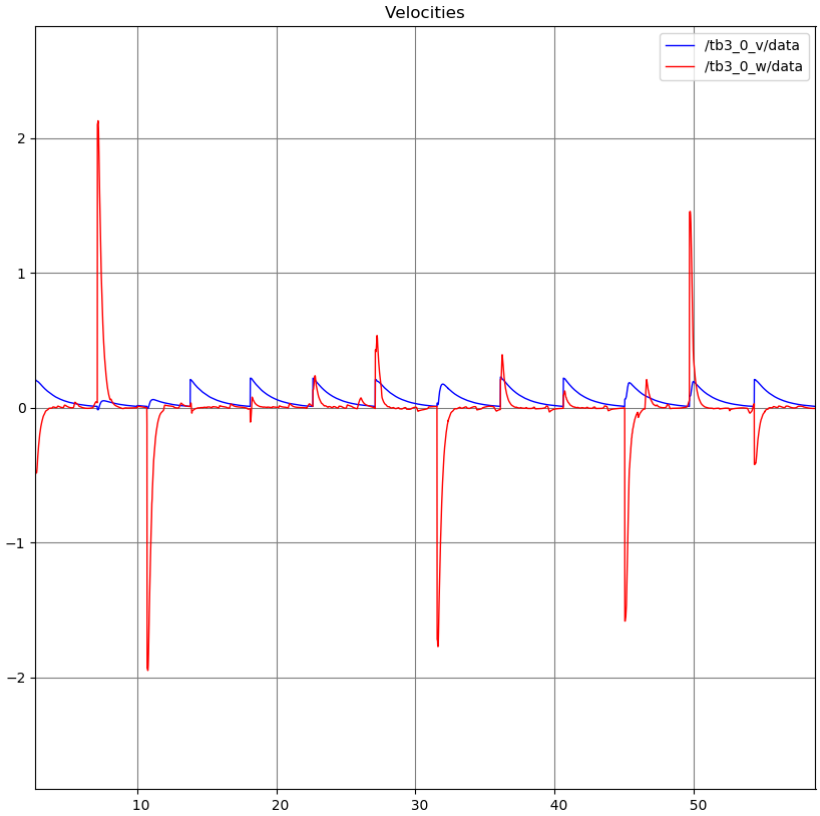


Figure 4.4: Physical inputs in Linear velocity (v) and angular velocity (ω)

4.1.2 I/O Feedback Linearization

This tracking controller significantly reduces the number of peaks in linear velocity and realizes a smoother trajectory in position(see Fig. 4.5 and 4.6) and linear velocity, with no guarantees on the sign of linear velocity.

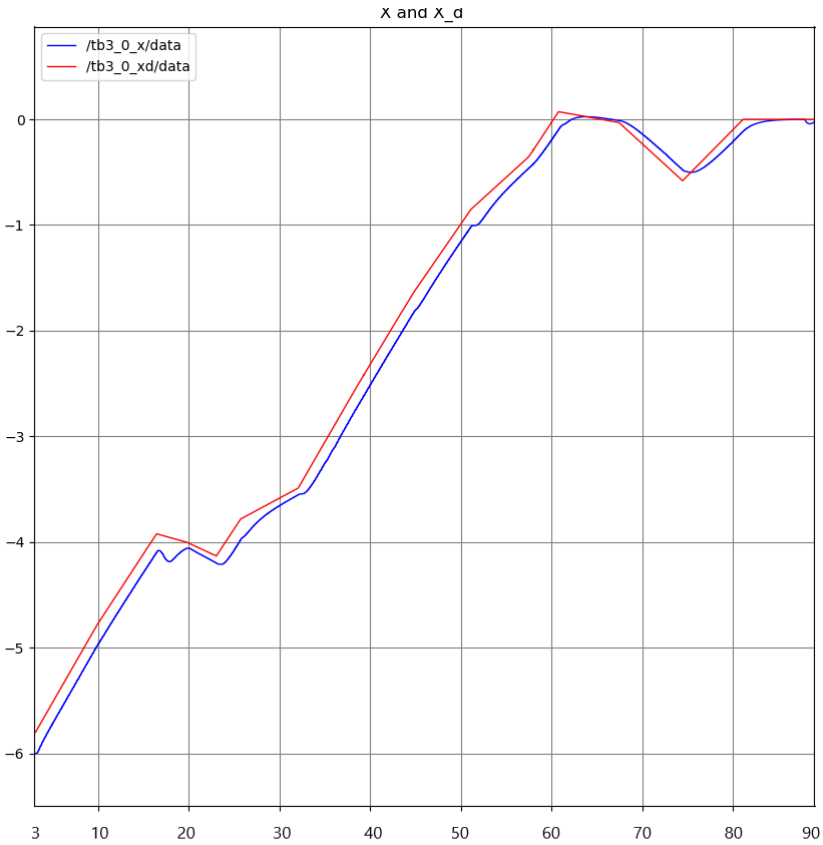


Figure 4.5: Evolution of coordinate x and x_d

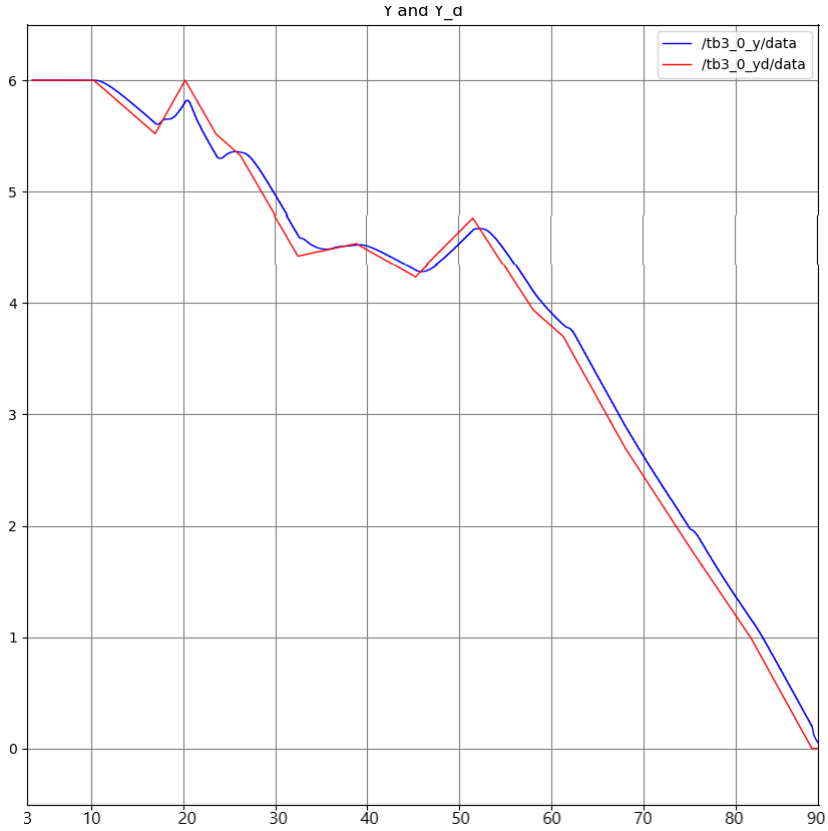


Figure 4.6: Evolution of coordinate y and y_d

It's worth noticing in Fig.4.7, between $15s$ and $25s$, a sudden change in the linear velocity which results in a delay and might result in a loss of stability of the robot, like slipping phenomena and/or central body oscillation, as the burger model has a higher structure than the other models. The last circumstance could be very rare to see since limit velocities are very low.

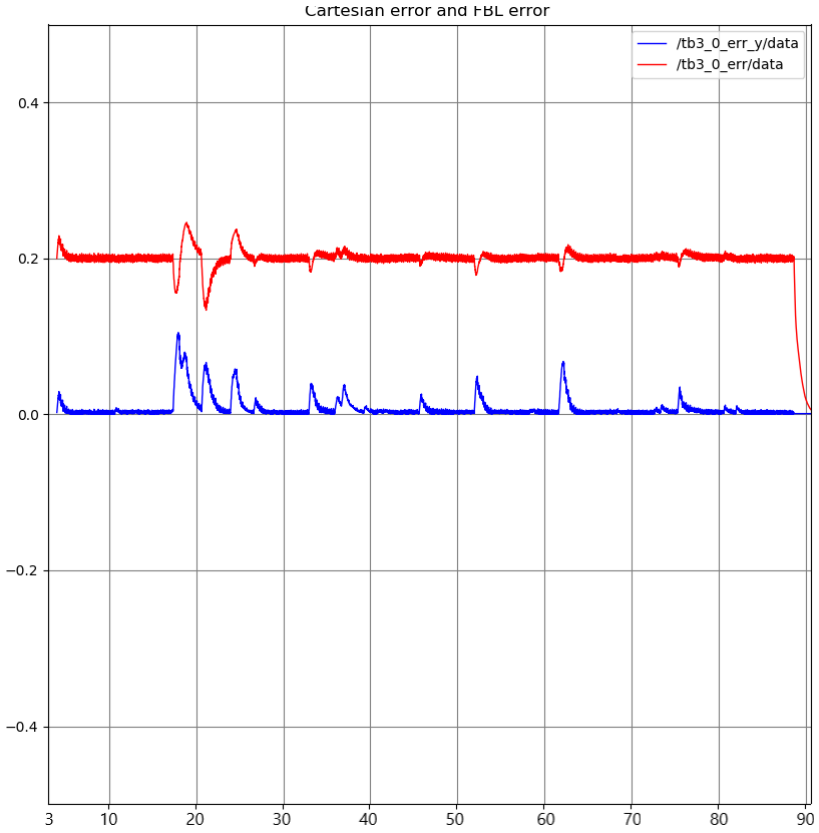


Figure 4.7: evolution of Cartesian error, in red, and error between $[y_1\, y_2]^T$ and $[y_{1,d}\, y_{2,d}]^T$, in blue.

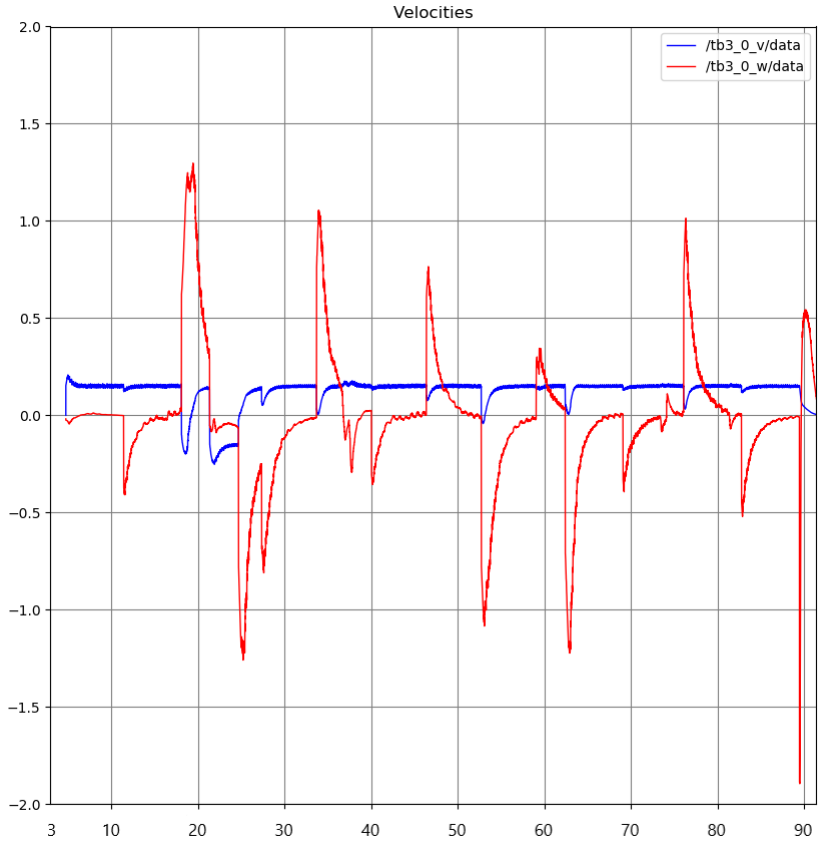


Figure 4.8: Physical inputs in Linear velocity (v) and angular velocity (ω)

As shown in Fig 4.7 *I/O FBL* tries to solve a tracking problem with respect to coordinates $[y_1\, y_2]^T$, i.e it's a tracking problem with respect to point B. Cartesian error suffers from a shift equal to the value of b which is reset to zero only with the help of a posture regulation controller, whose effect is obvious in the last part of the Fig. 4.7.

4.2 Conclusions

The project developed in this work shows that very good results can be achieved. Both the controllers reach an error value of less than 5 millimetres . This value strictly depends on sensors and map resolution and also on a good gain tuning of each controller.
Bidirectional-RRT achieves its goal as soon as it connects the two trees together. Even if this algorithm ensure an accurate solution(unlike from RRT whose stops whenever distance between goal and a connected node is less than ϵ) can lead to a real waste of time and therefore energy of the robot [6]. From simulation it appears that a very good result can be almost ensured extending the connection phase with multiple connection request, even if the starting trees are smaller, since they keep growind when connection fails. Besides a very important issue to consider is computational complexity, at this moment every algorithm requires to be run offline, and so does collision checking. Robot

does not take into account dynamic obstacle and obstacle avoidance in ensured only by configuration parameters like costmap parameters, controller gains and planned path accuracy.

Bibliography

[1] *AWS RoboMaker Bookstore World ROS package*. URL: https://github.com/aws-robotics/aws-robomaker-bookstore-world.

[2] *Comparison of Bidirectional-RRt with single or multiple connection*. URL: https://github.com/AndCav/FSR_PROJ.

[3] *ROSWiki: Gmapping*. URL: http://wiki.ros.org/gmapping.

[4] B. Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer, 2009.

[5] *Turtlebot Specifications*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/#specifications.

[6] *Comparison of Bidirectional-RRt with single or multiple connection*. URL: https://github.com/AndCav/FSR_PROJ/blob/master/single_multiple_connection.mp4.