Politecnico di Torino

Department of Electronics and Telecommunications

Master's degree in Mechatronic Engineering

2025/2026

Sensors, Embedded Systems and Algorithms for Service Robotics

Report 1

10/11/2025

Group n° 207

Marco Desogus – S346971

Paolo Ragusa – S344185

Siro Interlandi –S343898

Marco Conte – S346753

1 OBJECTIVE

The aim of this laboratory activity is to document the development and validation of a Bump and Go controller with the use of a ros2 node. The node was initially tested in a simulated environment using Gazebo and then on a real robot, following this workflow:

- Design of the ros2 package and node
- Simulation and testing in Gazebo
- Real robot experimental test

2 DESIGN

The Design part was mainly related to the creation of a ros2 node that was able to implement a Bump and Go algorithm, both in the simulation and the real robot. To do so, the node had to be built with python and ros2 paradigms; in this case the structure of the node was as follows:

- Node's Constructor
- The subscribers' methods
- Methods helpful for the main algorithm
- Publisher method (with inside the main control algorithm)

2.1 Node's Constructor

Inside the node's constructor are present the methods to create the publisher and subscriber (inherited from the Node class), also some attributes that are helpful to define the algorithm's characteristics:

Publisher

In this case it's a velocity command, it publishes the geometry_msgs/msg/Twist type message into the topic called /cmd_vel. The Twist message is a velocity message with inside the linear and angular values for the x, y, z axes.

Subscriber

There are two subscribers: one for the LiDAR data and one for the odometry data. For the LiDAR data the topic is called /scan and the message read inside is sensor_msgs/msg/LaserScan, for the odometry data the topic is called /odom and the message inside is nav_msgs/msg/Odometry.

Attributes

Some other particular attributes were added to the constructor as listed in Fig.1. The timer is needed for Ros to know at which frequency the node should operate, in this case a frequency of 5Hz (0.2 s) was chosen to have a suitable stream of data sent to the robot. Other parameters were defined to set linear and angular velocities and also some linear and angular thresholds for the robot to know when is too close to an obstacle and when the rotation he made was inside a defined tolerance.

```
# Timer: controls frequency of velocity updates
timer_period=0.2  # seconds
self.timer = self.create_timer(timer_period, self.velocity_callback)

#Initializing the laser data and odometry data
self.scan_info = None
self.yaw_odom_info = 0.0

#Parameters
self.linear_vel = 0.2                          # linear velocity at
self.angular_vel = 0.4                         # angular velocity a
self.front_dist_threshold = 0.2                # lowest distance va
self.yaw_tolerance = math.radians(3)           # tolerance for the
self.max_turn_angle = math.radians(45)         # the maximum angle

# State variables
#these variables are used to manage the rotation during time
self.rotating = False
self.target_yaw = None
```

Figure 1: Other attributes of the constructor

2.2 Subscriber's method

The subscriber's role is to read the data that is published on a certain topic. In the LaserScan case, the whole message was taken and put into the scan_info attribute that was initialized into the constructor's method. The odometry message instead was not of interest in the overall since it contains the whole robot pose information inside a quaternion object. The interest was only in the yaw angle, so with the euler_from_quaternion function, it was computed and put into the initialized attribute called yaw_odom_info.

```
#Subscriber callbacks
def laser_callback(self, msg: LaserScan):
    self.scan_info = msg

def odom_callback(self, msg: Odometry):
    quat = msg.pose.pose.orientation
    _, _, yaw = euler_from_quaternion([quat.x, quat.y, quat.z, quat.w])
    self.yaw_odom_info = yaw #we are interested only in the yaw information from the odometry
```

Figure 2: Subscriber's callback functions

2.3 Methods for the main algorithm

In this section there will be described two methods that will be used as functions in the main algorithm, the role of these methods is to make the code more modular and simpler to read from the outside. In general the objective here is to identify if the robot can go forward or not and if not, in which direction it can and should go.

The first method is called scan_front and its goal is to look at the heading of the robot and check if the forward path is clear or not by looking if the distance from an eventual obstacle. It takes as input the ranges vector, that is inside the LaserScan message, that has a dimension equal to the resolution of the LiDAR installed on the robot. In this case the whole circumference was divided into 360 elements so a resolution of 1°. It was chosen to control the 10° at the left and the 10° at the right of the heading, resulting in a method as shown in Fig. 3:

```
def scan_front(self, ranges):
    #This function checks 20° of the laser scan with the center at the front of the robot
    #and returns the minimum distance that detects in that sector
    front_sector = ranges[-10:] + ranges[:10]
    return min(front_sector)
```

Figure 3: scan_front method

The second method is called search_direction, also here the function takes as input the ranges vector. The main objective here is to:

- Look at the left and right of the robot

Two sectors are defined, the right_sector and the left_sector, both have a 45° window from the heading of the robot.

- Choose the maximum distance from any obstacle in both sides
- Evaluate which is the best
- Compute the angle and return the new target yaw that the robot will have to assume

The angle formula is a bit different based on which side it's chosen by the algorithm, this is because the ranges vector stores the measurements from the first angle (0°) that is the heading direction, in a counterclockwise rotation up to 360°. The left sector in this way is mapped from the first 45° of the ranges vector, the right one instead is mapped from the last 45 elements of the vector. To account this difference the two sides are computed differently:

$$angle = best\_index * angle\_increment \quad \text{for the left side}$$

$$angle = -((sector\_points - best\_index) * angle\_increment) \quad \text{for the right side}$$

where best_index is the index of the maximum distance chosen, angle_increment is the minimum angle given by the resolution of the LiDAR (1° in this case) and sector_points is the dimension of the right sector. The angle is then clamped between -45° and 45°, so the maximum rotation the robot can do is between those values, it's a design choice made to constrain the rotation so that the path taken is pretty much towards the forward direction. With other angles, such as 90° the maximum rotation could also direct the robot backwards.

The new yaw angle, called new_target_yaw is then computed by summing the current yaw, read from the odometry data, and the angle computed.

2.4 Publisher's method

Inside the publisher's method is present the main control algorithm that uses the function described above to make the robot move. A simple graph that shows how the control works is pictured in Fig.4:

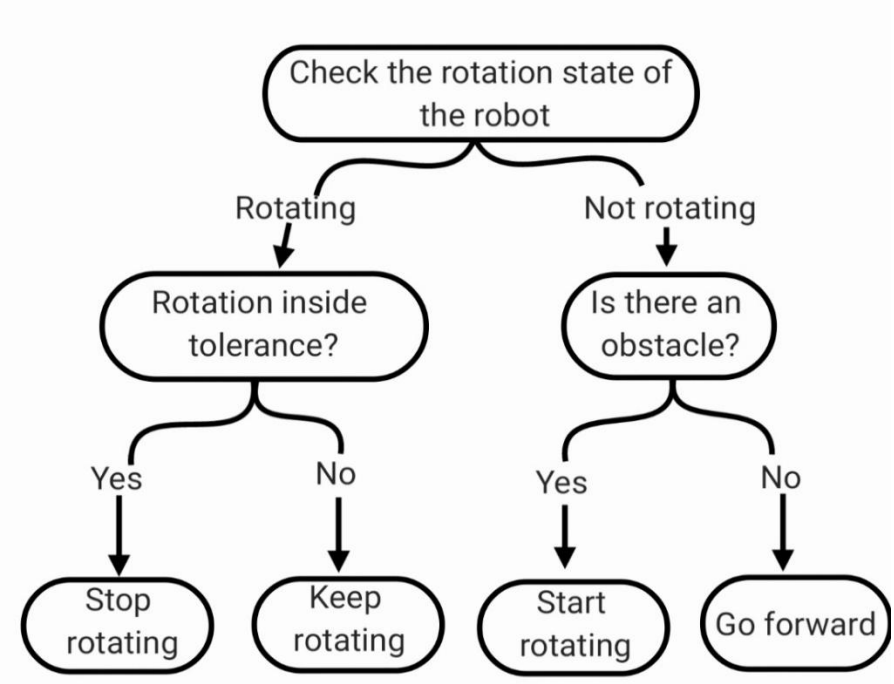

Figure 4: Graph of the main control algorithm

The algorithm mainly relies on a state, a particular attribute designed to keep memory of how the system in behaving. It's called rotating and if the state is False, the robot is not rotating otherwise the opposite.

This is done to remember the target yaw when the algorithm detects an obstacle, without this change of state during the rotation, the system would continue to update the target yaw every callback, entering in a not escapable loop and making the robot rotate continuously following a new target yaw.

In the not rotating branch, it goes forward if the scan_front returns a value that is higher than the threshold defined. Otherwise, it updates the state to rotating=True and computes the target yaw with the search_direction method, imposing also the z angular velocity.

During the robot rotation the algorithm enters the other branch and it checks if the rotation made is inside the defined tolerance. If it's not inside the tolerance it keeps rotating, otherwise the state gets reset to rotating=False and the angular velocity gets reset. A tolerance value is needed because it's very difficult for the system to reach the exact angle value, frequency of the callback, tolerance value and angular velocity, play all a role in the reaching of the desired angle. In fact, with a lower frequency, the system would not reach a bounded error but an oscillating one, making the robot continuously oscillate left and right from the target. To prevent this from occurring, the parameters were set in a suitable way. Another, more complex but also more robust, method would also implement a control system for the input angular velocity, so that when it's approaching the goal it would slow down, reducing the overshoot in the angle position.

3 EXPERIMENTS

The experiments were conducted first in simulation and then on the real robot, using the developed Bump and Go algorithm. The robot to be controlled is the TurtleBot3 burger, a two-wheeled differential robot that is equipped with a LiDAR, a microcontroller and an onboard PC running ROS2. The same configuration and same parameters were used on both the procedures, to accommodate the possible variation from the LiDAR defined in the simulation from the real one, the variables angle_max, angle_min or angle_increment were used, present in the LaserScan message instead of simply using the ranges indexes.

3.1 SIMULATION

The TurtleBot3 Burger was simulated in *Gazebo* using the official TurtleBot3 ROS 2 packages with a launch file on a map which contains walls to form a maze with a goal position, represented by a green cube (Figure 5). So, the map let the robot move forward until detecting an obstacle, choose a direction, rotate and continue exploring. The simulation has been run several times changing parameters such as the frontal threshold and the dimension of the left and right angular sector. For each run, the robot started from the same initial pose $(x, y, \theta) = (0, 0, 0)$, and the goal for the node controller was to reach the target or demonstrate a stable behaviour without collisions. At the beginning, the robot goes forward, because it is its natural movement, and then when it encounters an obstacle (a wall), it starts rotating to change the direction according to the algorithm.

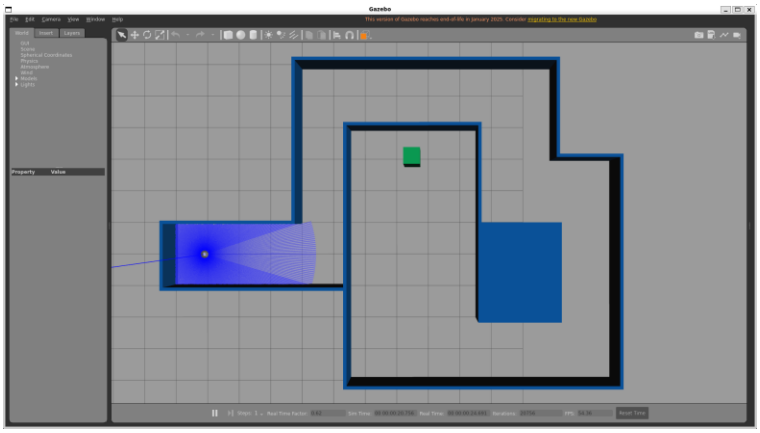

Figure 5: Gazebo environment

3.2 REAL ROBOT

After validating the controller in simulation, the same node was executed on the real TurtleBot3. The objective was to confirm that the algorithm behaved consistently in the real world when dealing with real sensor noise, friction or wheel slip. However, before executing the experiment, it is necessary to prepare the robot and establish the communication with its onboard computer. To do that, it is required to follow some steps. First, a proper Wi-Fi connection is needed, that is the two devices must be connected to the same network; the TurtleBot3 must be powered using a 11.1 V lithium battery for the motors and a 18V battery for the internal PC and then a ROS 2 connection is necessary to work on the same ROS domain.

The map of the simulation was replaced with cardboard walls to create a path as a simplified maze, but essential to test the controller's operation (Figure 6). Once connected to the robot via SSH, the bump-and-go node was launched and the experiment has been tried several times, due to differences between simulated and real sensing, that led to the tuning of the parameters, especially for the distance threshold and the linear velocity. In fact, at first the robot did not move because the maximum velocity used in the simulation was too high and the TurtleBot3 did not read it, so it was chosen equal to 0.15 m/s instead of the maximum value of 0.22 m/s. After some tests, the threshold was tuned to 0.2 m and the angular velocity to 0.4 rad/s to make the robot reaching the goal.

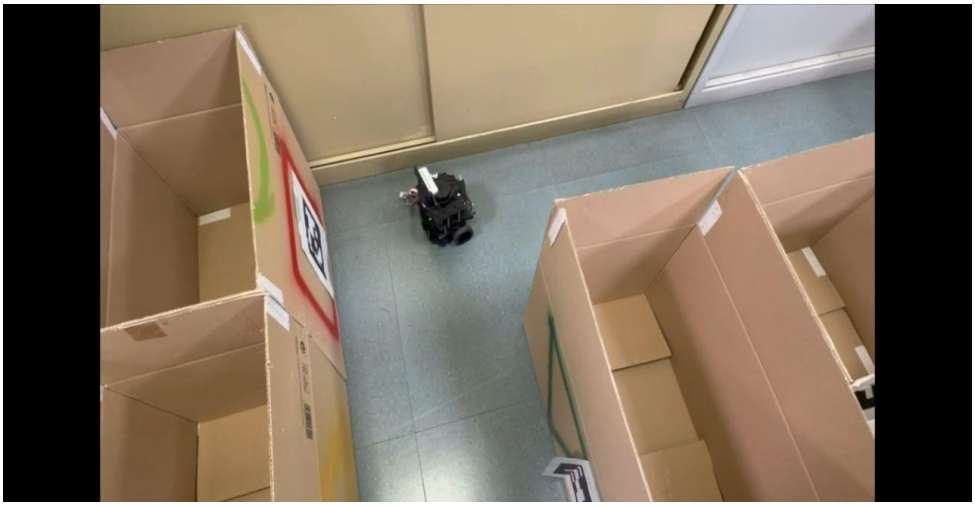

Figure 6: Setting in the real world

4 RESULTS AND DISCUSSION

The result of both simulation and real robot testing were then recorded with a rosbag callout during the execution. This allowed to save all the odometry and laser data used by the system during the motion and the results obtained are reported below. The most important data to be collected was the x-axis linear velocity and the z-axis angular velocity, both read from the /odom and the /cmd_vel topics. To analyse the information, the PlotJuggler software was used.

4.1 SIMULATION

The simulation data is plotted in the figures below:

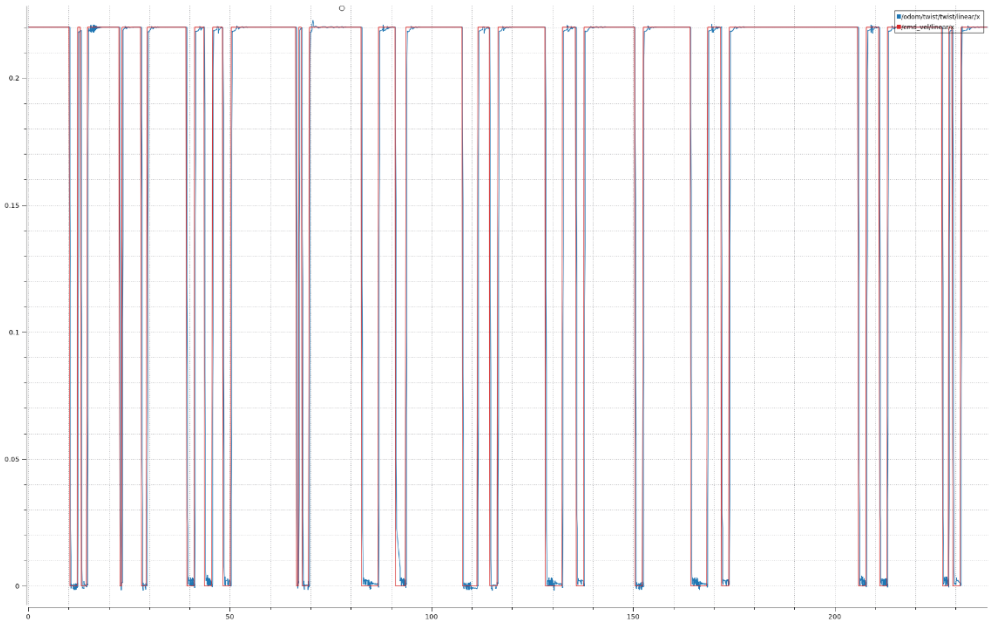

Figure 7: Linear x velocity

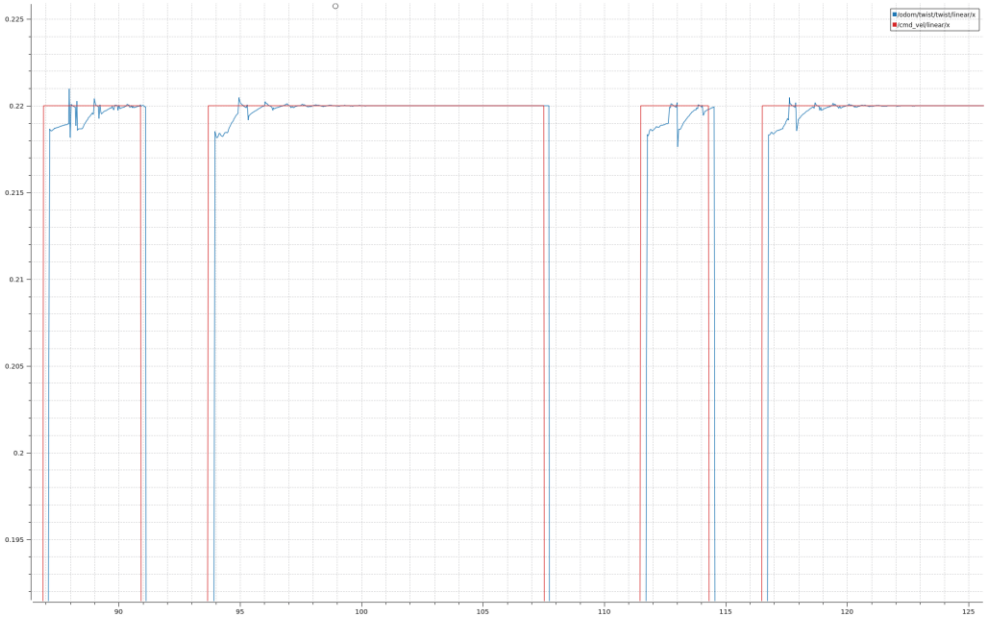

Figure 8: Zoom in the linear plot

Figure 7 shows the linear x velocity measured from the odometry data (blue line) and the publisher's velocity command. It's possible to see that in the simulation environment the robot follows almost exactly the command imposed apart from some Gaussian noise that is injected into the simulation. It's also possible to see that it's following the algorithm in a good way since the velocity has a step-like function, assuming the maximum speed when the robot has no obstacle in front and zero when it detects it and it must rotate. Fig. 8 is a zoom in of the previous figure and shows the dynamic response in the odometry with respect to the published input.

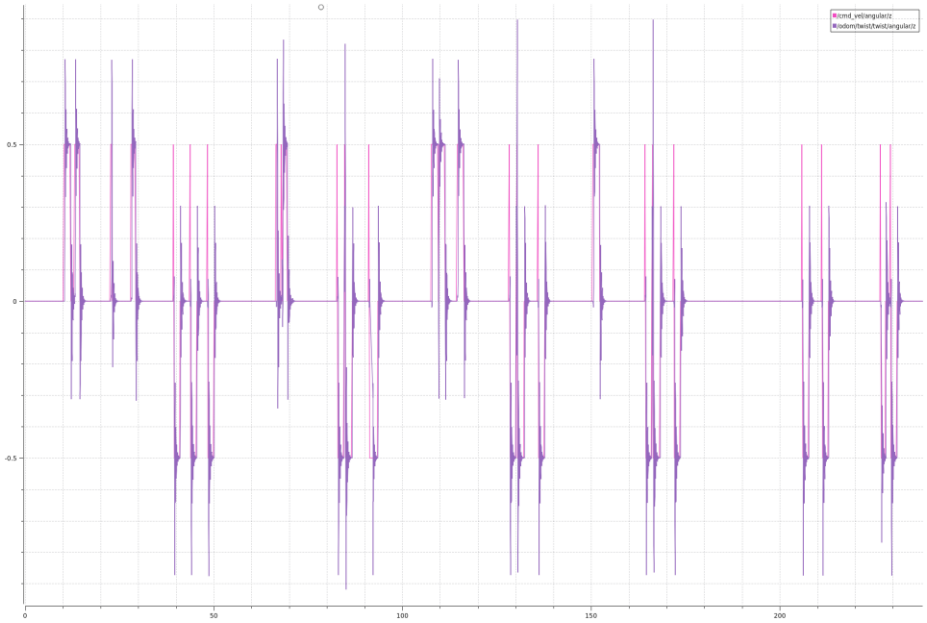

Fig. 9 shows instead the angular velocity comparison in the z-axis. The pink line is the published velocity and the purple one the odometry data. It's possible to see also here a dynamic response and some noise affecting the signal resulting in a slight difference between the two plots.

Figure 9: Angular velocities plot

4.2 REAL ROBOT

In the real robot experiment the same reasoning applies, with the results plotted below.

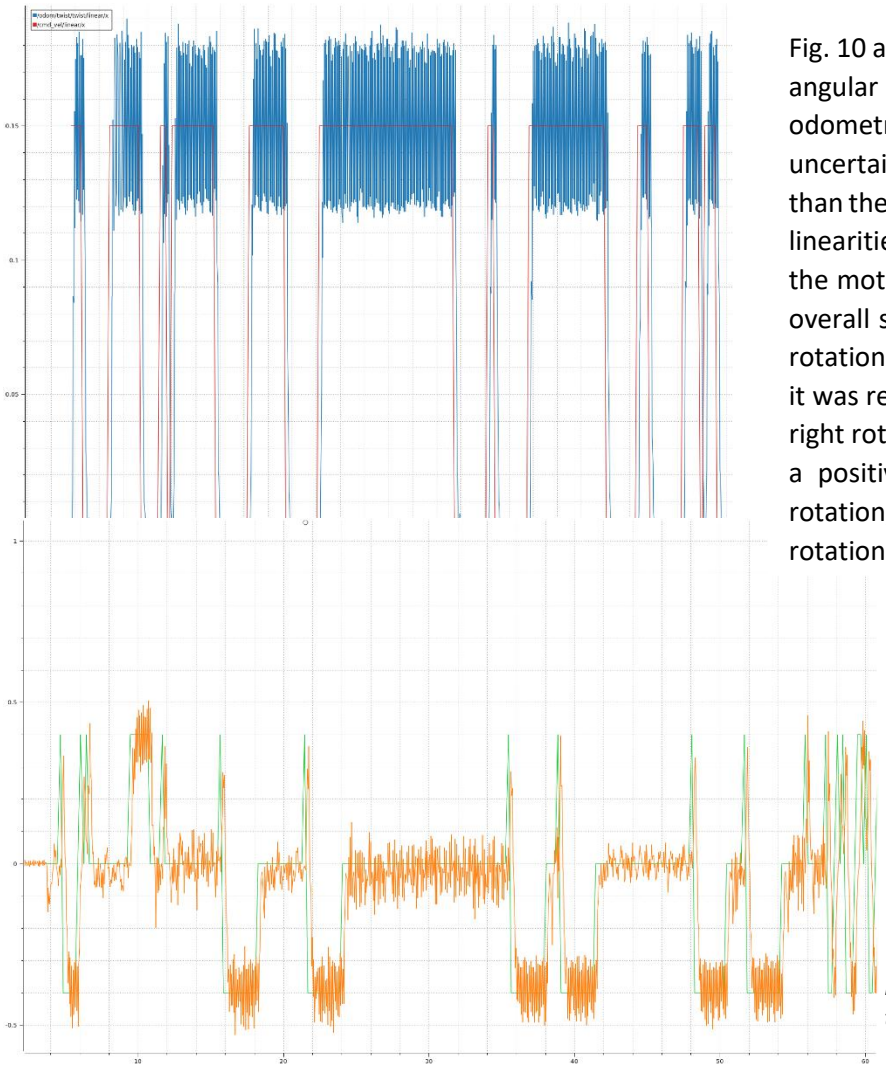

Fig. 10 and 11 depict, respectively, the linear and angular velocities of the command velocity and odometry data. It's possible to see here that the uncertainty is much larger in the odometry data than the simulated one, this is due to the real non linearities that affect the whole system such as the motor encoders and the overall noise in the overall system. It's also possible to see that the rotation is not as quick as in the simulation since it was reduced to a safer working value. Left and right rotation can be distinguished from the sign, a positive rotation means a counterclockwise rotation, a negative sign meant a clockwise rotation.

Figure 10-11: Linear and angular velocities in the real robot test

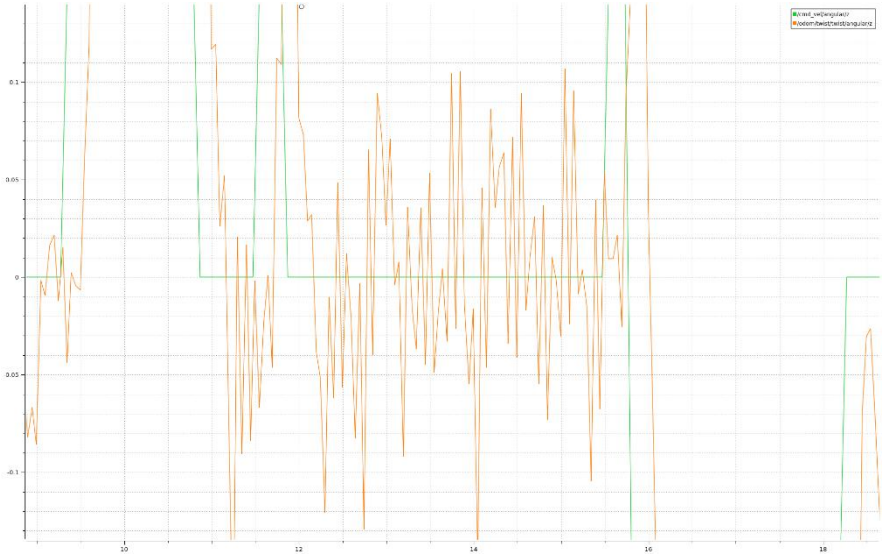

Fig.12: Zoom on the angular velocity plot, it's possible to see the noise affecting the odometry data with respect to the command velocity data.

Figure 12: Angular velocity noise

5 CONCLUSIONS

The implemented Bump and Go node successfully achieved the obstacle avoidance goal, both in simulation and in real testing. Differences in the two experiments were due to the noise model present, but the general behavior remained the same in both. Some improvements can be made to achieve smoother transitions from the linear motion and the angular one with the implementation of a controller to have variable velocities in order to slow down when the robot is approaching the goal pose.