A Backstepping Non-Smooth Controller for ROS-based Differential-Drive Mobile Robots

Walter Fetter Lages

Universidade Federal do Rio Grande do Sul,
Av. Osvaldo Aranha, 103,
90035-190 Porto Alegre RS, Brazil,
fetter@ece.ufrgs.br
http://www.ece.ufrgs.br/~fetter

Abstract. This chapter presents a non-linear controller for a mobile robot based on feedback linearization, non-smooth feedback and backstepping. The stability and convergence of the controller to the reference pose is proved by using the Lyapunov theory and the Barbalat Lemma. The controller design is based on a robot model considering its kinematics and dynamics, and hence the control inputs are the torques applied on the wheels. Contrariwise to most available implementation of controllers in the Robot Operating System, which implements a set of single input, single output controllers using the proportional + integral + derivative control law, here a truly multi-input, multi-output non-linear controller is considered. Results showing the effectiveness of the proposed controller for the setting point and the trajectory tracking problems were obtained by using the Gazebo robot simulator and Rviz.

Keywords: Backstepping, Multi-input multi-output, Non-linear Controller, Differential drive, Mobile robot

1 Introduction

This chapter proposes a control law for a differential-drive mobile robot which considers its dynamic model and not only the usual kinematic model. This control law is based on feedback linearization to handle the dynamics of the robot, non-smooth feedback to cope with the non-holonomicity of the kinematic model and backstepping to consider the cascade structure of the complete model (considering the kinematics and the dynamics effects) of the mobile robot.

The proposed controller is implemented on the Robot Operating System (ROS). Despite ROS being a widely used framework nowadays, its documentation and examples covering low-level controllers are poor and almost all implementations are based on single input, single output (SISO) controllers using the classical proportional + integral + derivative (PID) control law. Most references and textbooks on ROS [9, 21, 23, 26] do not even cover the implementation of controllers. On the other hand, robots are, in general, non-linear multi-input, multi-output systems (MIMO), for which the use of independent PID controllers for each degree of freedom (DoF) is not adequate if high performance is desired.

As the controller proposed in this work is non-linear and MIMO, its implementation in ROS is generic and representative of any controller and can be used as example for further implementation of any other control law (either linear or non-linear, SISO or MIMO, etc.), while the controllers available from standard ROS packages, such as `ros_controllers` are only representatives of SISO controllers. To capitalize on the generality of the implementation of the proposed controller, the inner working of the real-time loop of ROS is explained in detail, motivating the further implementation of advanced controllers in ROS such as in [18], where a computed torque controller is implemented to control a biped robot.

To verify the controller performance, the mobile robot is described in the Unified Robot Description Format (URDF) [21] and simulated by using the Gazebo simulator [11]. It is important to note that the robot model used for control purposes was derived independently from the model used by Gazebo to simulate the mobile robot. Furthermore, the robot is described in URDF in greater detail than the model used for control purposes considers. Hence, there is some model mismatch between the model used for simulation and the model used for control, as would be the case with a real robot, showing the robustness of the proposed approach. The stability and convergence of the closed-loop system is proved by using Lyapunov stability analysis and the Barbalat Lemma [25].

Similar concepts are used in the `lyap_control` ROS package [32]. However, its implementation is based on a ROS node and topics are used to communicate with sensors and actuators. This architecture is not real-time safe, as opposite to a controler based on the `ros_control` package as proposed here. This problem is even worse because a known drawback of the controller used in `lyap_control` is that it requires very small sampling periods, which are difficult to achieve without a proper real-time system. Furthermore, it requires the DoFs of the system to be the same as the number of its inputs, which does not hold for a differntial-drive mobile robot.

2 Background

In this section the background required to understand the implementation of the controller is presented. Section 2.1 presents the details of the model of a differential-drive mobile robot, including its dynamics and kinematics. That model is the base for the design of a backstepping controller presented in section 2.2. The implemantion of that controller requires information about the robot pose, which for the purpose of this chapter is computed by the odometry procedure described in section 2.3. Of course, as any odometry-only based pose estimation, it is subject to drift and for more ambitious applications, a proper pose estimation with sensor fusion should be used. Nonetheless, the procedure described in 2.3 can be used as the odometry sensor for that more sophisticades sensor fusion based pose estimation.

2.1 Mobile Robot Model

The model of the mobile robot used for control purposes is described in this section. Figure 1 shows the coordinate systems used to describe the mobile robot model, where X_c and Y_c are the axes of the coordinate system attached to the robot and X_0 and Y_0 form the inertial coordinate system. The pose (position and orientation) of the robot is represented by $\boldsymbol{x} = \begin{bmatrix} x_c & y_c & \theta_c \end{bmatrix}^T$.

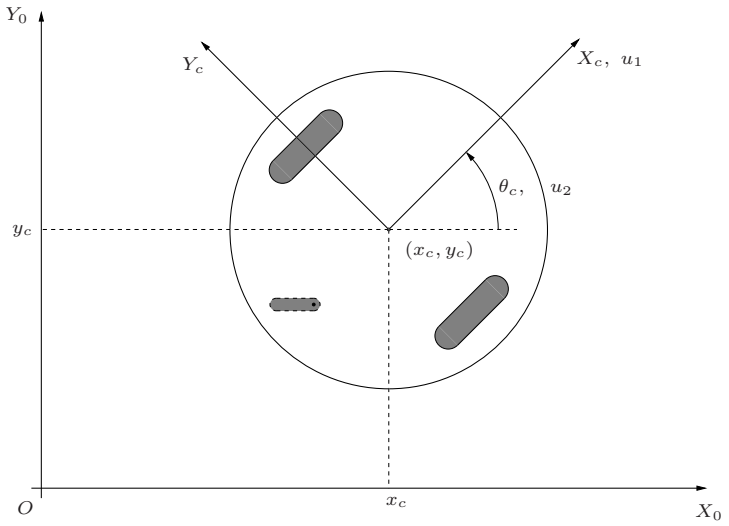

Fig. 1. Coordinate systems.

The dynamic model of the TWIL mobile robot used in this work can be obtained based on the Lagrange-Euler formulation [6] and is given by [14]:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{B}(\boldsymbol{x})\boldsymbol{u} \\ \dot{\boldsymbol{u}} &= \boldsymbol{f}(\boldsymbol{u}) + \boldsymbol{G}\boldsymbol{\tau} \end{aligned} \tag{1}$$

where $\boldsymbol{u}$ is the vector of the linear and angular velocities of the robot and $\boldsymbol{\tau}$ is the vector of input torques on the wheels. $\boldsymbol{B}(\boldsymbol{x})$ is a matrix whose structure depends on the kinematic (geometric) properties of the robot, $\boldsymbol{f}(\boldsymbol{u})$ and $\boldsymbol{G}$ depend on the kinematic and dynamic (mass and inertia) parameters of the robot and for the TWIL mobile robot are given by:

$$\boldsymbol{B}(\boldsymbol{x}) = \begin{bmatrix} \cos\theta_c & 0 \\ \sin\theta_c & 0 \\ 0 & 1 \end{bmatrix} \tag{2}$$

$$\boldsymbol{f}(\boldsymbol{u}) = \begin{bmatrix} 0 & f_{12} \\ f_{21} & 0 \end{bmatrix} \begin{bmatrix} u_1 u_2 \\ u_2^2 \end{bmatrix} = \boldsymbol{F} \begin{bmatrix} u_1 u_2 \\ u_2^2 \end{bmatrix} \tag{3}$$

$$\boldsymbol{G} = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{21} \end{bmatrix} \tag{4}$$

where $f_{12} = 0.08444$ m/rad^2, $f_{21} = 3.7706$ m^{-1}, $g_{11} = g_{12} = 2.6468$ kg^{-1}m^{-1} and $g_{21} = -g_{22} = -16.0840$ kg^{-1}m^{-2} are constants depending only on the geometric and inertia parameters of the robot.

Note that the dynamic model of the robot is a cascade between its kinematic model (the first expression of (1)), which considers velocities as inputs, and its dynamics (the second expression of (1)), which considers torques as inputs, as shown in Figure 2.

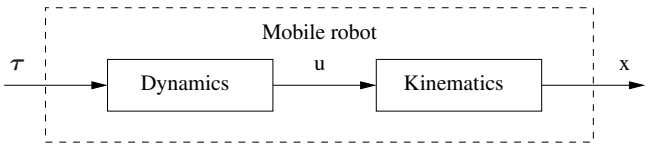

Fig. 2. Cascade between dynamics and the kinematic model.

2.2 Control of the Mobile Robot

Differential-drive mobile robots are nonholonomic systems [6]. An important general statement on the control of nonholonomic systems has been made by [4], who has shown that it is not possible to asymptotically stabilize the system at an arbitrary point through a time-invariant, smooth state feedback law. In spite of it, the system is controllable [2].

Ways around Brockett's conditions for asymptotic stability are time-variant control [24, 31, 8, 27], non-smooth control [2, 29, 7] and hybrid control laws [17]. Most of those control laws considers only the kinematics of the mobile robot and hence only the first equation of (1), neglecting the dynamics effects described by the second equation of (1). Nonetheless, due to the cascading structure of (1), it is possible to use a control law designed to control only the kinematics of the mobile robot to design a control law for the whole model (1) in a procedure called backstepping [12].

Here, for the kinematic part of the robot model, a set of possible input signals is obtained by using a non-smooth coordinate transform. A general way of

designing control laws for nonholonomic systems through non-smooth coordinate transform was presented by [2]. Here, we consider a non-smooth coordinate transform similar to [15], which was already implemented in ROS [14].

Before the design of the non-smooth control law and the backstepping, two other steps are performed: a feedback linearization [10, 28] to further simplify the model of the mobile robot and a coordinate change in order to enable the steering of the robot to any reference pose and not only to the origin.

Feedback Linearization By using the feedback [14]:

$$\boldsymbol{\tau} = \boldsymbol{G}^{-1}\left(\boldsymbol{v} - \boldsymbol{f}(\boldsymbol{u})\right) \tag{5}$$

where $\boldsymbol{v}$ is a new input vector, it is possible linearize the second expression of (1) to obtain:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{B}(\boldsymbol{x})\boldsymbol{u} \\ \dot{\boldsymbol{u}} &= \boldsymbol{v} \end{aligned} \tag{6}$$

Note that the model (6) still has a cascade structure, but with a much simpler dynamics with accelerations instead of torque as input. That cascade structure enables the design of a backstepping controller.

Offset to Origin In the following sections, a controller for (6) is designed under the assumption that the robot should converge to the origin. However, it is interesting to stabilize the robot at any pose $\boldsymbol{x}_r = \begin{bmatrix} x_{cr} & y_{cr} & \theta_{cr} \end{bmatrix}^T$. This can be done by a coordinate change [15] given by (see Figure 3):

$$\bar{\boldsymbol{x}} = \begin{bmatrix} \cos\theta_{cr} & \sin\theta_{cr} & 0 \\ -\sin\theta_{cr} & \cos\theta_{cr} & 0 \\ 0 & 0 & 1 \end{bmatrix} (\boldsymbol{x} - \boldsymbol{x_r}) \tag{7}$$

Non-Smooth Control In this section, a non-smooth control law for stabilizing the kinematics of the mobile robot is presented without details. See [14] for the details of how this control law is obtained. Also, see [16] for coordinate changes for details on control laws for mobile robot not based on differential-drive kinematics.

By considering a coordinate change [3], which is similar to a change to polar coordinates,

$$e = \sqrt{\bar{x}_1^2 + \bar{x}_2^2} \tag{8}$$

$$\psi = \operatorname{atan2}(\bar{x}_2, \bar{x}_1) \tag{9}$$

$$\alpha = \bar{x}_3 - \psi \tag{10}$$

$$\eta_1 = u_1 \tag{11}$$

$$\eta_2 = u_2 \tag{12}$$

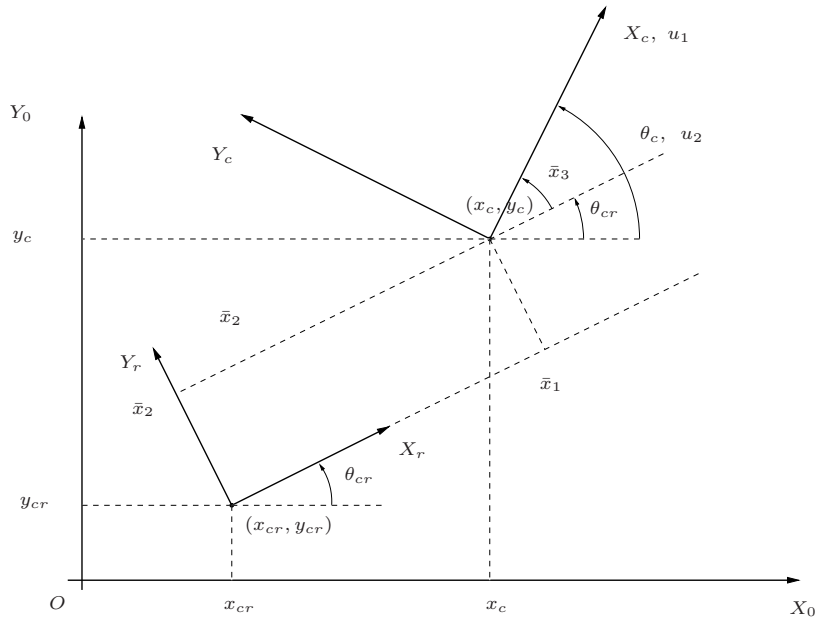

Fig. 3. Robot coordinates with respect to the reference frame.

the first expression of system model (6) can be rewritten as:

$$\begin{aligned}\dot{e} &= \cos\alpha\eta_1 \\ \dot{\psi} &= \frac{\sin\alpha}{e}\eta_1 \\ \dot{\alpha} &= -\frac{\sin\alpha}{e}\eta_1 + \eta_2.\end{aligned} \tag{13}$$

which is forced to converge to the origin by the following input signal [1]:

$$\eta_1 = -\gamma_1 e\cos\alpha \tag{14}$$

$$\eta_2 = -\gamma_2\alpha - \gamma_1\cos\alpha\sin\alpha + \gamma_1\frac{\lambda_3}{\lambda_2}\cos\alpha\frac{\sin\alpha}{\alpha}\psi \tag{15}$$

with $\lambda_i > 0$ and $\gamma_i > 0$.

Backstepping Although (14-15) are able to stabilize the first equation of (6), they can not stabilize (6) as a whole since its input is $\boldsymbol{v}$ and not $\boldsymbol{u}$. Note, however that (6) can be seen as a cascade between two subsystems and in this case, it is possible to apply a backstepping procedure [12] to $\boldsymbol{u}$ in order to obtain an expression for $\boldsymbol{v}$.

By applying the transforms (7, 8-10) to (6) it is possible to write:

$$\dot{e} = \cos\alpha u_1 \tag{16}$$
$$\dot{\psi} = \frac{\sin\alpha}{e} u_1 \tag{17}$$
$$\dot{\alpha} = -\frac{\sin\alpha}{e} u_1 + u_2 \tag{18}$$
$$\dot{u}_1 = v_1 \tag{19}$$
$$\dot{u}_2 = v_2 \tag{20}$$

Then, by adding $\cos\alpha(\eta_1 - \eta_1)$ to (16), $\frac{\sin\alpha}{e}(\eta_1 - \eta_1)$ to (17) and $-\frac{\sin\alpha}{e}(\eta_1 - \eta_1) + (\eta_2 - \eta_2)$ to (18) and rearranging it is possible to write:

$$\begin{aligned}
\dot{e} &= \cos\alpha\eta_1 + \cos\alpha(u_1 - \eta_1) \\
\dot{\psi} &= \frac{\sin\alpha}{e}\eta_1 + \frac{\sin\alpha}{e}(u_1 - \eta_1) \\
\dot{\alpha} &= -\frac{\sin\alpha}{e}\eta_1 + \eta_2 - \frac{\sin\alpha}{e}(u_1 - \eta_1) + (u_2 - \eta_2) \\
\dot{u}_1 &= v_1 \\
\dot{u}_2 &= v_2
\end{aligned} \tag{21}$$

and by defining:

$$e_1 \triangleq u_1 - \eta_1 \tag{22}$$
$$e_2 \triangleq u_2 - \eta_2 \tag{23}$$
$$\bar{v}_1 \triangleq v_1 - \dot{\eta}_1 \tag{24}$$
$$\bar{v}_2 \triangleq v_2 - \dot{\eta}_2 \tag{25}$$

results in:

$$\begin{aligned}
\dot{e} &= \cos\alpha\eta_1 + \cos\alpha e_1 \\
\dot{\psi} &= \frac{\sin\alpha}{e}\eta_1 + \frac{\sin\alpha}{e}e_1 \\
\dot{\alpha} &= -\frac{\sin\alpha}{e}\eta_1 + \eta_2 - \frac{\sin\alpha}{e}e_1 + e_2 \\
\dot{e}_1 &= \bar{v}_1 \\
\dot{e}_2 &= \bar{v}_2
\end{aligned} \tag{26}$$

Then, by replacing η_1 and η_2 from (14) and (15):

$$\begin{aligned}
\dot{e} &= -\gamma_1 e\cos^2\alpha + \cos\alpha e_1 \\
\dot{\psi} &= -\gamma_1\sin\alpha\cos\alpha + \frac{\sin\alpha}{e}e_1 \\
\dot{\alpha} &= -\gamma_2\alpha + \gamma_1\frac{\lambda_3}{\lambda_2}\cos\alpha\frac{\sin\alpha}{\alpha}\psi - \frac{\sin\alpha}{e}e_1 + e_2 \\
\dot{e}_1 &= \bar{v}_1 \\
\dot{e}_2 &= \bar{v}_2
\end{aligned} \tag{27}$$

Let the following candidate to Lyapunov function:

$$V_1 = \frac{1}{2}\left(\lambda_1 e^2 + \lambda_2 \alpha^2 + \lambda_3 \psi^2 + \lambda_4 e_1^2 + \lambda_5 e_2^2\right) \tag{28}$$

which, by differentiating with respect to time and replacing the system equations from (27) gives:

$$\begin{aligned}\dot{V}_1 = &-\gamma_1 \lambda_1 e^2 \cos^2\alpha - \gamma_2 \lambda_2 \alpha^2 + \lambda_1 e \cos\alpha e_1 \\ &- \lambda_2 \alpha \frac{\sin\alpha}{e} e_1 + \lambda_3 \psi \frac{\sin\alpha}{e} e_1 \\ &+ \lambda_4 e_1 \bar{v}_1 + \lambda_2 \alpha e_2 + \lambda_5 e_2 \bar{v}_2\end{aligned} \tag{29}$$

Then, by choosing:

$$\bar{v}_1 \triangleq -\gamma_3 e_1 - \frac{\lambda_1}{\lambda_4} e \cos\alpha + \frac{\lambda_2}{\lambda_4} \alpha \frac{\sin\alpha}{e} - \frac{\lambda_3}{\lambda_4} \psi \frac{\sin\alpha}{e} \tag{30}$$

$$\bar{v}_2 \triangleq -\gamma_4 e_2 - \frac{\lambda_2}{\lambda_5} \alpha \tag{31}$$

results in:

$$\dot{V}_1 = -\gamma_1 \lambda_1 e^2 \cos^2\alpha - \gamma_2 \lambda_2 \alpha^2 - \gamma_3 \lambda_4 e_1^2 - \gamma_4 \lambda_5 e_2^2 \leq 0 \tag{32}$$

which proves that V_1 is indeed a Lyapunov function for the system (27). Then, it follows that e, α, e_1 and e_2 are bounded, which implies that $\ddot{V}_1$ is bounded as well, which ensures that $\dot{V}_1$ is uniformly continuous. Furthermore, since $\dot{V}_1$ is uniformly continuous, it follows from the Barbalat lemma [25, 28] that $\dot{V}_1 \to 0$, which implies that $e \to 0$, $\alpha \to 0$, $e_1 \to 0$ and $e_2 \to 0$. It remains to prove that ϕ converges to zero. By applying the Barbalat's lemma to $\dot{\alpha}$ it follows that $\dot{\alpha} \to 0$ in (27), which implies that $\psi \to 0$.

The control law for the system (6) is then, from (24-25) given by:

$$v_1 = \bar{v}_1 - \dot{\eta}_1 \tag{33}$$

$$v_2 = \bar{v}_2 - \dot{\eta}_2 \tag{34}$$

A block diagram of the proposed control schema is presented in Figure 4.

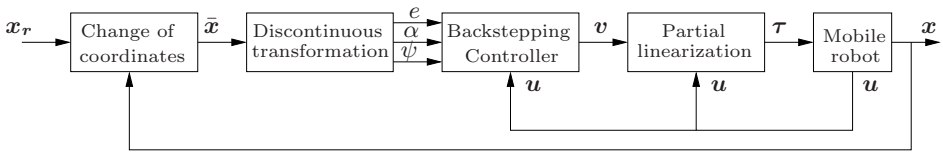

Fig. 4. Block diagram of the proposed controller.

2.3 Odometry

For a differential-drive mobile robot, all factible trajectories are composed of arcs of circumferences. The straight line is just a circumference with infinite radius. Hence, it makes more sense to compute the odometry by supposing an arc of circumference between adjacent sampling points. This is more precise than supposing an straight line between sampling points as it is usually done. Therefore, as shown in Figure 5, the the pose estimate $\hat{\boldsymbol{x}}_o$ is given by [22]:

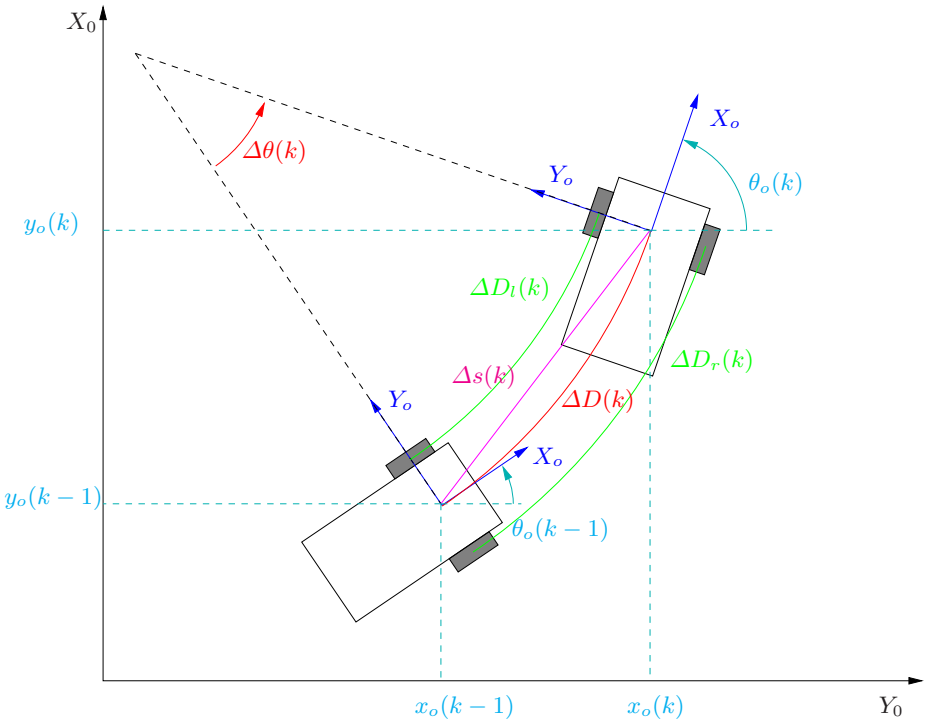

Fig. 5. Odometry supposing an arc of circumference trajectory.

$$\hat{\boldsymbol{x}}_o(k) = \hat{\boldsymbol{x}}_o(k-1) + \Delta\hat{\boldsymbol{x}}_o(k) \tag{35}$$

with

$$\Delta\hat{\boldsymbol{x}}_o(k) = \begin{bmatrix} \Delta s(k)\cos\left(\theta(k) + \frac{\Delta\theta(k)}{2}\right) \\ \Delta s(k)\sin\left(\theta(k) + \frac{\Delta\theta(k)}{2}\right) \\ \Delta\theta(k) \end{bmatrix} \tag{36}$$

$$\Delta s(k) = \Delta D(k) \frac{\sin\left(\frac{\Delta\theta(k)}{2}\right)}{\frac{\Delta\theta(k)}{2}} \tag{37}$$

and

$$\begin{bmatrix} \Delta D(k) \\ \Delta\theta(k) \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\left(\Delta\varphi_r(k) r_r + \Delta\varphi_l(k) r_l\right) \\ \frac{1}{2b}\left(\Delta\varphi_r(k) r_r - \Delta\varphi_l(k) r_l\right) \end{bmatrix} = \Delta \boldsymbol{U}(k) \tag{38}$$

where $\Delta\varphi_r(k)$ and $\Delta\varphi_l(k)$ are the angular displacement of the right and left wheels, respectively, r_r and r_l are the radii of the right and left wheels, respectively and $2b$ is the axial distance between the right and left wheels..

Note that usually odometry is computed by assuming a straight line trajectory between two successive points, while in (35-38) an arc of circumference is assumed, which is the actual path of the robot if there is no error in kinematics parameters and the wheels do not slip.

3 ROS Setup

This section describes the installation of some packages useful for the implementation of the backstepping controller described in Section 2. Some of them are not present in a standard installation of ROS and should be installed. Also, some custom packages with our implementation of the TWIL robot model and the backstepping controller should be installed. Most of those packages are already described in detail in [13] and/or in [14] and hence, it will not be repeated here. Just the installation instructions for those packages will be given.

3.1 Setting up a Catkin Workspace

The packages to be installed for implementing ROS controllers assume an existing catkin workspace. If it does not exist, it can be created with the following commands (assuming a ROS Indigo version):

```
source /opt/ros/indigo/setup.bash
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

3.2 `ros_control`

The `ros_control` meta-package includes a set of packages to implement generic controllers. It is not included in the standard ROS desktop installation, hence it should be installed. On Ubuntu, it can be installed from Debian packages with the command:

```
sudo apt-get install ros-indigo-ros-control
```

3.3 ros_controllers

This meta-package implements a set of controllers to be used in ROS. In particular, the `joint_state_controller` controller, is used to obtain the output of the robot system. This meta-package is not included in the standard ROS desktop installation, hence it should be installed. On Ubuntu, it can be installed from Debian packages with the command:

```
sudo apt-get install ros-indigo-ros-controllers
```

3.4 gazebo_ros_pkgs

This is a collection of ROS packages for integrating the `ros_control` controller architecture with the Gazebo simulator [11] and is not included in the standard ROS desktop installation, hence it should be installed. On Ubuntu, it can be installed from Debian packages with the command:

```
sudo apt-get install ros-indigo-gazebo-ros-pkgs ros-indigo-gazebo-ros-control
```

3.5 twil

This is a meta-package with the package with the description of the TWIL robot. It contains an URDF description of the TWIL mobile robot. More specifically it includes the following packages:

`twil_description`: URDF description of the TWIL mobile robot.
`twil_ident`: identification of the dynamic parameters of the TWIL mobile robot. This package is not used here and was descripted in [14].

The `twil` meta-package can be downloaded and installed in the ROS catkin workspace with the commands:

```
cd ~/catkin_ws/src
wget http://www.ece.ufrgs.br/twil/indigo-twil-20180303.tgz
tar -xzf indigo-twil-20180303.tgz
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

3.6 arc_odometry

This package implements an odometry procedure supposing that the robot trajectory between two adjacent sampling points is an arc of circunference, as described in section 2.3. Its implementation is described in section 5.2 and it can be downloaded and installed in the ROS catkin workspace with the commands:

```
cd ~/catkin_ws/src
wget http://www.ece.ufrgs.br/ros-pkgs/indigo-arc-odometry-20180303.tgz
tar -xzf indigo-arc-odometry-20180303.tgz
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

3.7 pose2d_trajectories

This package implements simple trajectory generators for testing the controllers. It can be downloaded and installed in the ROS catkin workspace with the commands:

```
cd ~/catkin_ws/src
wget http://www.ece.ufrgs.br/ros-pkgs/indigo-pose2d-trajectories-20180303.tgz
tar -xzf indigo-pose2d-trajectories-20180303.tgz
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

3.8 nonsmooth_backstep_controller

This packake has the implementation of the backstepping controller. The background for the theory of operation of the controller is described in section 2, while its implementation is discussed in section 5. It can be downloaded and installed in the ROS catkin workspace with the commands:

```
cd ~/catkin_ws/src
wget http://www.ece.ufrgs.br/ros-pkgs/indigo-nonsmooth-backstep-controller-20180303.tgz
tar -xzf indigo-nonsmooth-backstep-controller-20180303.tgz
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

4 Testing the Installed Packages

A simple test for the installation of the packages described in section 3 is performed here.

The installation of the ROS packages can be done by loading the TWIL model in Gazebo and launching the controller with the commands:

```
source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
roslaunch nonsmooth_backstep_controller gazebo.launch
```

The robot should appear in Gazebo as shown in Figure 6.

Then, start the simulation by clicking in the play button in the Gazebo panel, open a new terminal and issue the following commands to move the robot.

```
source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
rosrun nonsmooth_backstep_controller pose_step.sh 5 0 0
```

The `pose_step.sh` script publishes the setting point pose for the controller. In this case, $x_c = 5$, $y_c = 0$ and $\theta = 0$. Hence the robot should move to a point 5 m ahead of its current position.

If everything is right, the TWIL robot should move for some seconds and then stop, as shown in Figure 7.

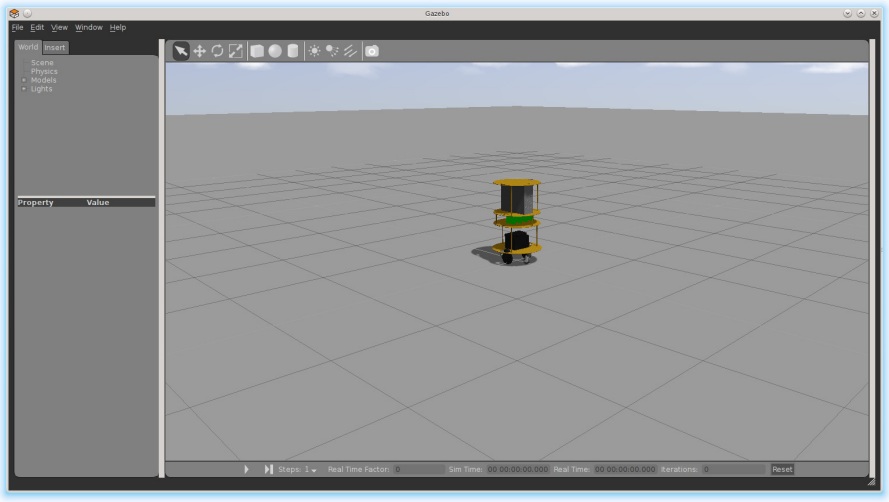

Fig. 6. TWIL mobile Robot in Gazebo.

Note that the trajectory of the robot is not necessary a straight line. The trajectory is implicit determined by the controller and its parameters. If a specific trajectory to reach a final point is desired, then, the reference to the controller should not be the final point itself, but a sequence of points produced by a trajectory generator. See the example in section 6 for an 8 trajectory.

By checking the `/nonsmooth_backstep_controller/status` topic with the command:

```
rostopic echo /nonsmooth_backstep_controller/status
```

it is possible to verify that the robot pose (the `process_value` field in the topic) converges to the desired pose.

In this simulation, the TWIL mobile robot is driven by the nonsmooth backstepping controller, which receives a message of the `geometry_msgs/Pose2D` type and forces the robot to converge to the 2D pose it describes.

The `pose_step.sh` is a script with an example of how to publish the reference for the nonsmooth backstepping controller, which is message of the `geometry_msgs/Pose2D` type. The script just publishes the required values by using the `rostopic` command. In a real application, that references would be generated by a planning package, such as MoveIt! [30] or a robot navigation package, such as the Navigation Stack [19, 20].

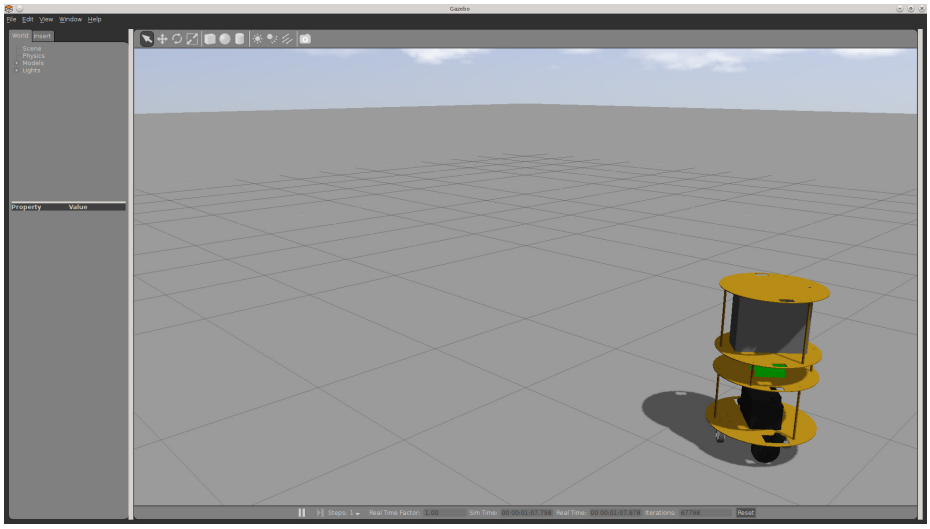

Fig. 7. Gazebo with TWIL robot after a step test motion.

5 Implementation of the Backstepping Controller in ROS

This section describes the details of the implementation of the controller described in section 2.2 and some other ROS packages which are used by the examples.

5.1 The `twil_description` Package

The `twil_description` package has the URDF description of the TWIL mobile robot and is covered in detail in [14]. Hence, it will not be discussed further here.

5.2 The `arc_odometry` Package

The `arc_odometry` package, shown if Figure 8 implements the odometry procedure described in section 2.3. Th package contains a library, called `libarc_odometry.so` and a ROS node, called `odometry_publisher`. The purpose of the library is to enable the reuse of the code in the implementation of other packages, while the node enables the standalone use of the arc odometry with already existing packages.

The library is implemented in the `diff_odometry.h` and `diff_odometry.cpp` files, which define the `DiffOdometry` class, shown in Listing 1. The `update()` function computes the odometry, as described by (35)-(38). For convenience of use, there are two overloaded versions of the function: one taking as argument the angular displacement on each wheel and another one taking as arguments a vector with both angular displacements. The other functions are used to get the

```
arc_odometry/
├── CMakeLists.txt
├── config/
│   ├── odometry_publisher.yaml -> twil.yaml
│   └── twil.yaml
├── include/
│   └── arc_odometry/
│       └── diff_odometry.h
├── launch/
│   └── gazebo8.launch
├── package.xml
└── src/
    ├── diff_odometry.cpp
    └── odometry_publisher.cpp
```

Fig. 8. `arc_odometry` package.

pose and velocity of the robot as a vector or as individual components and to set the parameters of the odometry and initial pose.

Listing 1. `DiffOdometry` class.

```
        class DiffOdometry
        {
                public:
                DiffOdometry(double wheelBase,std::vector<double> wheelRadius);
                ~DiffOdometry(void);

                void update(double leftDisp,double rightDisp,const ros::Duration &duration);
                void update(const Eigen::Vector2d &deltaPhi,const ros::Duration &duration);
                double x(void) const {return x_[0];}
                double y(void) const {return x_[1];}
                double heading(void) const {return x_[2];}
                void getPose(Eigen::Vector3d &x) const {x=x_;}
                double linear(void) const {return u_[0];}
                double angular(void) const {return u_[1];}
                void getVelocity(Eigen::Vector2d &u) const {u=u_;}
                void setParams(double wheelBase,std::vector<double> wheelRadius);
                void setPose(const Eigen::Vector3d &x) {x_=x;}

                private:
                std::vector<double>wheelRadius_;
                double wheelBase_;

                Eigen::Vector2d u_;
                Eigen::Vector3d x_;
        };
}
```

By using the library, the `odometry_publisher` ROS node is implemented in the `odometry_publisher.cpp` file. It subscribes to the `joint_states` topic to obtain the joint positions, which are used to compute the joint displacements and the odometry and publishes the `odom` and `/tf` topics, as shown in Figure 9.

The parameters for the `odometry_publisher` ROS nodes are configured in the `config/odometry_publisher.yaml` file. In the default package it is a sym-

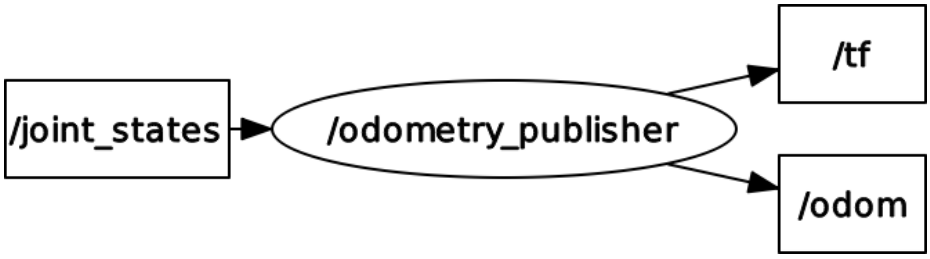

Fig. 9. Topics used by the `odometry_publisher`.

bolic link to the `twil.yaml`, which is configured with the values for the TWIL robot.

In this chapter, the `odometry_publisher` node is not used. The implementation of the non-smooth backstepping controller uses directly the library as it makes simpler to synchronize dht pose and velocity estimateswot the controller updates.

5.3 The `pose2d_trajectories` Package

The purpose of the `pose2d_trajectories` package is to implement nodes for generating the messages with the reference trajectories to be followed by the controllers implemented in the `nonsmooth_backstep_controller` package. It is not a full featured trajectory generator, but just a collection of simple trajectory publishers to be used to test the controllers. The reference for the nonsmooth backstepping controller is a `geometry_msg:Pose2D` message but the Rviz tool, in its default configuration, can not show this type of message, but can show `geometry_msgs:PoseStamped`, this package also implements a node to convert `geometry_msgs:Pose2D` messages in `geometry_msgs:PoseStamped` messages.

5.4 The `nosmooth_backstep_controller` Package

The `nonsmooth_backstep_controller` package, shown in figure 10, implements the controller proposed in section 2.2. The files in the `config` directory specify the parameters for the controller, such as the joints of the robot associated to the controller, its gains and sampling rate. The files in the `src` directory are the implementation of the controller described in section 2.2 (the `NonSmoothBackstepController` class), while the `include` directory holds the files with the declaration of that class. The `nonsmooth_backstep_controller_plugins.xml` file specifies that the class implementing the controller is a pluginsfor the ROS controller manager. The files in the `launch` directory are used to launch (i.e. load) the controller with the respective configuration files. The messages defined in the `msg` directory are used for publishing the status of the nonsmooth backstepping controller.

```
nonsmooth_backstep_controller/
├── CMakeLists.txt
├── config/
│   ├── nonsmooth_backstep_control.yaml -> twil.yaml
│   ├── twil.yaml
│   └── twil_step.yaml
├── include/
│   ├── nonsmooth_backstep_controller/
│   │   └── nonsmooth_backstep_controller.h
├── launch/
│   ├── adaptive_nonsmooth_backstep.launch
│   ├── gazebo.launch
│   ├── gazebo8.launch
│   └── nonsmooth_backstep.launch
├── msg/
│   ├── NonSmoothBackstepControllerStatus.msg
│   └── PosePolar.msg
├── package.xml
├── scripts/
│   ├── pose_step.py
│   └── pose_step.sh
├── src/
│   ├── nonsmooth_backstep_controller.cpp
└── nonsmooth_backstep_controller_plugins.xml
```

Fig. 10. `nonsmooth_backstep_controller` package.

The `effort_controllers/NonSmoothBackstepController` is the implementation of the controller described in section 2.2. It is a MIMO controller, which receives the desired pose of the robot as its reference and computes the torques to apply to each wheel of the robot. Figure 11 shows the topics subscribed and published by the controller.

The reference to the controller is received through the `/nonsmooth_backstep_controller/command` topic, and actuates the robot wheels by calling some member functions of the `JointHandle` class. See [13] for details.

Three other topics are also published by the controller: the `/nonsmooth_backstep_controller/status` topic, which exposes many internal variables of the controller, the `/nonsmooth_backstep_controller/odom` topic, where the odometry data computed by the controller for its internal use is published and the `/tf` topic, where the transformation of the base frame of the mobile robot with respect to the `odom` frame is published. The name of the base frame of the mobile robot is configured in the `.yaml` file in the `config` directory.

The `/nonsmooth_backstep_controller/status` topic can be used for debugging and tuning of the controller parameters while the `/nonsmooth_backstep_`

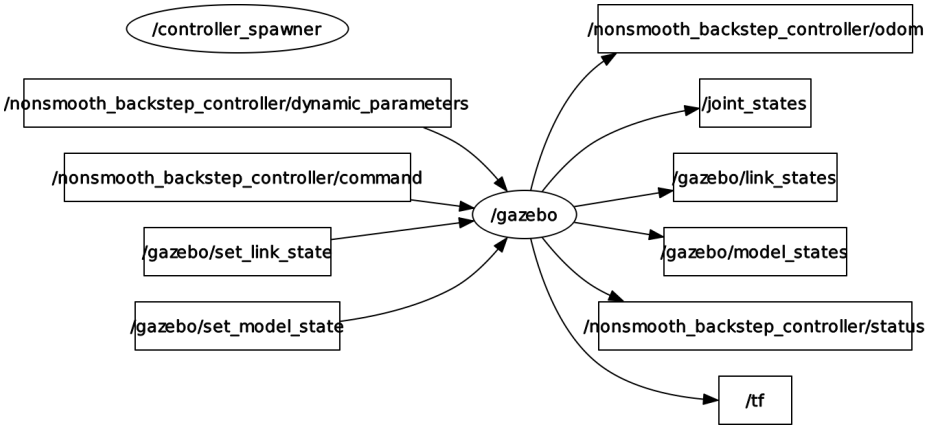

Fig. 11. Topics used by the nonsmooth backstepping controller.

`controller/odom` topic can be useful for other nodes in the system estimating the pose of the robot through data fusion.

The `/nonsmooth_backstep_controller/dynamic_parameters` topic is used to receive updates to the parameters of the dynamic model (3, 4), in an adaptive controller similar to the one described in details in [13], but this feature is not used here.

Listing 2 shows the `NonSmoothBackstepController` class. It is a typical controller using the `EffortJointInterface`. The public members are the constructor and the destructor of the class and the functions called to load the controller (`init()`), to start it (`starting()`) and to update the controller at each sampling time (`update()`).

Listing 2. `NonsmoothBackstepController` class.

```
class NonSmoothBackstepController: public controller_interface::
        Controller<hardware_interface::EffortJointInterface>
{
        public:
        NonSmoothBackstepController(void);
        ~NonSmoothBackstepController(void);

        bool init(hardware_interface::EffortJointInterface *robot,
                        ros::NodeHandle &n);
        void starting(const ros::Time& time);
        void update(const ros::Time& time,const ros::Duration& duration);

        private:
        ros::NodeHandle node_;
        hardware_interface::EffortJointInterface *robot_;
        std::vector<hardware_interface::JointHandle> joints_;

        boost::scoped_ptr<realtime_tools::RealtimePublisher
                <nonsmooth_backstep_controller::NonSmoothBackstepControllerStatus>
                >  status_publisher_ ;

        boost::shared_ptr<realtime_tools::RealtimePublisher
```

```
                    <nav_msgs::Odometry> > odom_publisher_;
            boost::shared_ptr<realtime_tools::RealtimePublisher
                    <tf::tfMessage> > tf_odom_publisher_;

            ros::Subscriber sub_command_;
            ros::Subscriber sub_parameters_;

            Eigen::Matrix2d Ginv_;
            Eigen::Matrix2d F_;

            arc_odometry::DiffOdometry odom_;

            Eigen::Vector3d xRef_;

            Eigen::Vector2d eta_;

            double time_step_;
            ros::Time lastSamplingTime_;

            Eigen::Vector2d phi_;

            std::vector<double> lambda_;
            std::vector<double> gamma_;

            void commandCB(const geometry_msgs::Pose2D::ConstPtr &command);
            void parametersCB(const std_msgs::Float64MultiArray::ConstPtr &command);
    };
```

The class has many handles which are private members: for the node, for the `EffortJointInterface` and for the joints, handles for publishing and subscribing topics and many internal variables and callback functions for the subscribed topics. All those are the usual for any ROS node. However, the handles for publishing require special attention: They are not the usual ROS publisher handles. These are not real-time safe and then can not be used from the update loop of a real-time controller. To overcome this problem, there is the `realtime_tools` package, which implements wrappers for the usual ROS publishers. The wrapped version of the publisher can be used in a realtime loop. The wrapper implements this feature by creating non-real-time threads to handle the message publishing.

The implementation of the constructor and destructor functions is trivial. they just call the constructors of member objects and their destructors, respectively. The `init()` and `starting()` functions, called on controller load and starting are simple, as well. They just initialize variables and load controller parameters from the parameter server.

More interesting things are done in the `update()` function, shown in Listing 3. This function is called by the controller manager at a sampling rate defined in the ⟨`controlPeriod`⟩ tag in the URDF description of the robot. See details in the `twil_description` package. The controller manager uses this same rate for all enable controller in the system. However, while that sampling period may be good for some controllers, it may be inadequate for others. Hence, if a specific controller should be run at a slower sampling rate, it should perform a downsampling to its required sampling rate. This is done by letting its `update()` function to implement a downsampling. The actual sampling period of the nonsmooth backstepping controller is configured by the `time_step` parameter and loaded in the parameter server though the `.yaml` file in the `config` directory.

Listing 3. `NonsmoothBackstepController::update()` function.

```
void NonSmoothBackstepController::update(const ros::Time& time,
        const ros::Duration& duration)
{
        ros::Duration dt=time-lastSamplingTime_;

        if(fabs(dt.toSec()-time_step_) > time_step_/20) return;
        lastSamplingTime_=time;

        // Incremental encoders sense angular displacement and
        // not velocity
        // phi[0] is the left wheel angular displacement
        // phi[1] is the right wheel angular displacement
        Eigen::Vector2d deltaPhi=-phi_;
        for(unsigned int i=0;i < joints_.size();i++)
        {
                phi_[i]=joints_[i].getPosition();
        }
        deltaPhi+=phi_;

        odom_.update(deltaPhi,dt);

        Eigen::Vector3d x;
        odom_.getPose(x);

        Eigen::Vector2d u;
        odom_.getVelocity(u);

        // Change of coordinates
        Eigen::Matrix3d R;
        R << cos(xRef_[2]), sin(xRef_[2]), 0.0,
                -sin(xRef_[2]), cos(xRef_[2]), 0.0,
                0.0, 0.0, 1.0;
        Eigen::Vector3d xBar=R*(x-xRef_);

        // Discontinuous transformation
        double e=sqrt(sqr(xBar[0])+sqr(xBar[1]));
        double psi=atan2(xBar[1],xBar[0]);
        double alpha=xBar[2]-psi;

        // deta=(eta(k)-eta(k-1)/dt
        Eigen::Vector2d deta=-eta_;

        // Backstepping
        eta_[0]=-gamma_[0]*e*cos(alpha);

        if(fabs(alpha ) > DBL_EPSILON) eta_[1]=-gamma_[1]*alpha
                -gamma_[0]*sin(alpha)*cos(alpha)+gamma_[0]*lambda_[2]*psi*sin(alpha)/
                lambda_[1]/alpha*cos(alpha);
        else eta_[1]=gamma_[0]*lambda_[2]*psi/lambda_[1];
        deta+=eta_;
        deta/=dt.toSec();

        Eigen::Vector2d eb=u-eta_;

        Eigen::Vector2d vBar;
        if(fabs(e) > DBL_EPSILON) vBar[0]=-gamma_[2]*eb[0]
                -lambda_[0]/lambda_[3]*e*cos(alpha)
                +lambda_[1]/lambda_[3]*alpha*sin(alpha)/e
               -lambda_[2]/lambda_[3]*psi*sin(alpha)/e;
        else vBar[0]=-gamma_[2]*eb[0]-lambda_[0]/lambda_[3]*e*cos(alpha);
        vBar[1]=-gamma_[3]*eb[1]-lambda_[1]/lambda_[4]*alpha;

        Eigen::Vector2d v=vBar+deta;

        // Linearization
        Eigen::Vector2d uf(u[0]*u[1],sqr(u[1]));
```

```
Eigen::Vector2d torque=Ginv_*(v-F_*uf);

// Apply torques
for(unsigned int i=0;i < joints_.size();i++)
{
        joints_[i].setCommand(torque[i]);
}

if(status_publisher_ && status_publisher_->trylock())
{
        status_publisher_->msg_.header.stamp=time;

        status_publisher_->msg_.set_point.x=xRef_[0];
        status_publisher_->msg_.set_point.y=xRef_[1];
        status_publisher_->msg_.set_point.theta=xRef_[2];

        status_publisher_->msg_.process_value.x=x[0];
        status_publisher_->msg_.process_value.y=x[1];
        status_publisher_->msg_.process_value.theta=x[2];

        status_publisher_->msg_.process_value_dot.x=u[0]*cos(x[2]);
        status_publisher_->msg_.process_value_dot.y=u[0]*sin(x[2]);
        status_publisher_->msg_.process_value_dot.theta=u[1];

        status_publisher_->msg_.error.x=xRef_[0]-x[0];
        status_publisher_->msg_.error.y=xRef_[1]-x[1];
        status_publisher_->msg_.error.theta=xRef_[2]-x[2];

        status_publisher_->msg_.time_step=dt.toSec();

        for(int i=0;i < torque.size();i++)
                status_publisher_->msg_.command[i]=torque[i];

        for(int i=0;i < lambda_.size();i++)
                status_publisher_->msg_.lambda[i]=lambda_[i];

        for(int i=0;i < gamma_.size();i++)
                status_publisher_->msg_.gamma[i]=gamma_[i];

        status_publisher_->msg_.polar_error.range=e;
        status_publisher_->msg_.polar_error.angle=psi;
        status_publisher_->msg_.polar_error.orientation=alpha;

        for(int i=0;i < eta_.size();i++)
                status_publisher_->msg_.backstep_set_point[i]=eta_[i];

        for(int i=0;i < deta.size();i++)
                status_publisher_->msg_.backstep_set_point_dot[i]=deta[i];

        for(int i=0;i < u.size();i++)
                status_publisher_->msg_.backstep_process_value[i]=u[i];

        for(int i=0;i < eb.size();i++)
                status_publisher_->msg_.backstep_error[i]=eb[i];

        for(int i=0;i < vBar.size();i++)
                status_publisher_->msg_.backstep_command[i]=vBar[i];

        for(int i=0;i < v.size();i++)
                status_publisher_->msg_.linear_dynamics_command[i]=v[i];

        status_publisher_->unlockAndPublish();
}

if(odom_publisher_ && odom_publisher_->trylock())
{
        odom_publisher_->msg_.header.stamp=time;
```

```
                odom_publisher_->msg_.pose.pose.position.x=x[0];
                odom_publisher_->msg_.pose.pose.position.y=x[1];
                odom_publisher_->msg_.pose.pose.orientation.z=sin(x[2]/2);
                odom_publisher_->msg_.pose.pose.orientation.w=cos(x[2]/2);

                odom_publisher_->msg_.twist.twist.linear.x=u[0]*cos(x[2]);
                odom_publisher_->msg_.twist.twist.linear.y=u[0]*sin(x[2]);
                odom_publisher_->msg_.twist.twist.angular.z=u[1];

                odom_publisher_->unlockAndPublish();
        }

        if(tf_odom_publisher_ && tf_odom_publisher_->trylock())
        {
                geometry_msgs::TransformStamped &odom_frame=
                        tf_odom_publisher_->msg_.transforms[0];
                odom_frame.header.stamp=time;
                odom_frame.transform.translation.x=x[0];
                odom_frame.transform.translation.y=x[1];
                odom_frame.transform.rotation.z=sin(x[2]/2);
                odom_frame.transform.rotation.w=cos(x[2]/2);

                tf_odom_publisher_->unlockAndPublish();
        }
}
```

Here, the backstepping controller should run with a sampling period of 10 ms, as configured by the `time_step` parameter in the `.yaml` file in the `config` directory. If the `update()` function is called in a shorter interval, it just returns.

Then, the incremental encoders mounted at the robot wheels are read with a call to the `getPosition()` function. Note that incremental encoders read angular displacement (and not velocity), which divided by the interval of time measured from the former read gives the average velocity within the interval. If the `getVelocity()` function were used instead, the result would be the instantaneous velocity at the sampling time, which, for the purpose of computing the odometry is less appropriate than the average velocity due to the noise. The `arc_odometry` library (see section 5.2) is used at this point to compute the robot pose and velocity.

In the sequence, the offset to origin, described in section 2.2 and the non-smooth coordinate change (8-10) are computed. The reference for the controller, used in the offset to origin is received through the `command` topic, as seen in Figure 11. It is a `geometry_msgs/Pose2D` message. Then, the backstepping control is computed, resulting in the virtual input for the linearized system, v.

With the virtual input for the linearized system, it is possible to compute the torques to be applied to the robot, by using (5). The values for the parameters G and F (used to compute $f(u)$) were identified through experiments with the actual robot, following the method described in [14], and are used to initialize the associated variables in the `init()` function of the controller. The default values are read from the `.yaml` file in the `config` directory, but they can be changed by publishing their values and the associated covariances to the `dynamic_parameters` topic, as can be seen in Figure 11. That is useful for implementing adaptive control versions of the controller.

The torque is then applied to the robot by calling the `setCommand()` function for each joint, concluding the control cycle. The remainder of the `update()`

function is devoted to publish the information produced by the controller. However, as the controller code should be real-time safe, the publishers are not the usual ROS publishers, but wrapped real-time versions, as discussed in section 5.4, which implies that the publication should be protected by mutexes [5]. This is done by calling the function `trylock()` before accessing the data fields of the message and then publishing it using the `unlockAndPublish()` function, instead of the usual function for publishing messages used in ROS nodes.

6 Results

This section presents the results of simulations of the TWIL robot in Gazebo using the backstepping controller, proposed in previous sections. Two types of references are used. First, the setting point problem, also known as the parking problem is considered. In this problem, a given target pose is applied to controler input and the robot should converge to this pose. This type of reference is used to evaluate the controller transient reponse. Note that a trajectory to drive the robot from initial pose to the target pose is not specified and is implicit determined by the controller parameters (λ_i and γ_i), but the controller ensures the convergence to the specified position and orientation.

If a specific trajectory is desired, then the trajetory tracking problem should be considered. In this problem, the robot should converge to a reference trajectory and follow it. Here, it is important to note that the actual trajectory of the robot is subject to the non-holonomic constraints. For a differential-drive robot, that means that its orientation should be aligned with the tangent of its trajectory in $X_0 \times Y_0$ plane. And this may be different from the orientation specified in the reference trajectory, even for a well planed factible trajectory, due to noise. Then, the controller attempt to force to robot to follow the reference position and the infactible orientation may produce some very unnatural trajectory following behavior. The solution to this problem is to tune the controller parameters (λ_i and γ_i) to permit a larger error in orientation. Then, the robot would be able to follow the reference position and align itself with the tangent to the actual trajectory, while allowing a larger orientation error if necessary.

From the discussion above, it should be clear that the set of controller parameters, in particular those related to orientation (λ_2 and λ_3) which are good for the setting point problem are not good for the trajectory tracking problem and vice-versa, as in the first problem the orientation should converge to the reference, and in the second problem some error should be allowed to enable the robot to align with the tangent to the actual trajectory. Here, the values were chosen as $\lambda_1 = 200.0$, $\lambda_2 = 6.0$, $\lambda_3 = 6.0$, $\lambda_4 = 500.0$, $\lambda_5 = 10000.0$, $\gamma_1 = 10$, $\gamma_2 = 1.0$, $\gamma_3 = 10.0$ and $\gamma_4 = 50.0$, which give an average performance for both cases. Of course, if better convergence of the orientation in the setting point problem is desired, the values of λ_2 and λ_3 can be increased, but then the preformance in the trajectory tracking would deteriorate.

6.1 Setting Point

For the setting point tests, the controller is loaded with the command:

```
roslaunch nonsmooth_backstep_controller gazebo.launch
```

.
and then, the `pose_step.sh` script can be used to supply a setting point to the controller, as done in section 4,

The Gazebo simulator is started in paused mode so that the initial condition for the simulation can be checked. By pressing the play button, the simulation starts.

Step in X_0 Direction In this case, the reference is set to $x_c = 1$, $y_c = 0$ and $\theta = 0$, by issuing the command:

```
rosrun nonsmooth_backstep_controller pose_step.sh 1 0 0
```

Figure 12 shows the Cartesian position of the robot while following the step reference in X_0 direction. In this figure it is possible to see the offset at the initial point of the trajectory and the robot convergence to the trajectory. Note that the tracking error is small and in part due to the pose estimation based on odometry.

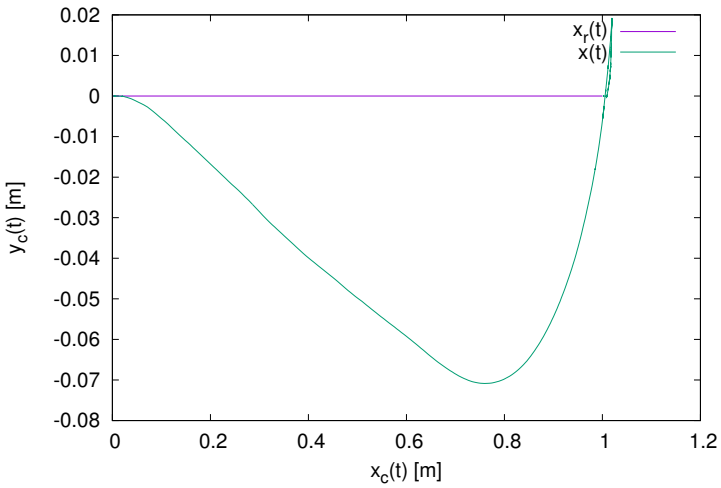

Fig. 12. Cartesian position $y_c(t) \times x_c(t)$ for a step reference in X_0 direction.

Figures 13, 14 and 15 show the reference pose and the robot pose over the time. Those figures show the transitories of $x_c(t)$, $y_c(t)$ and $\theta_c(t)$ while they converge to the reference values.

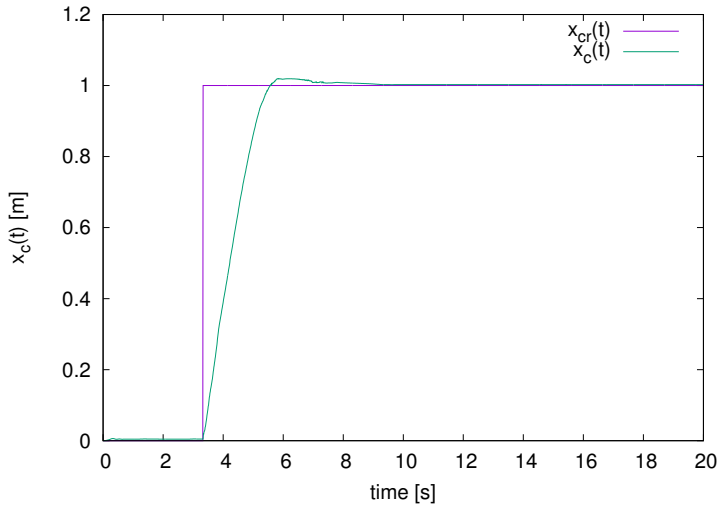

Fig. 13. Cartesian position $x_c(t) \times t$ for a step reference in X_0 direction.

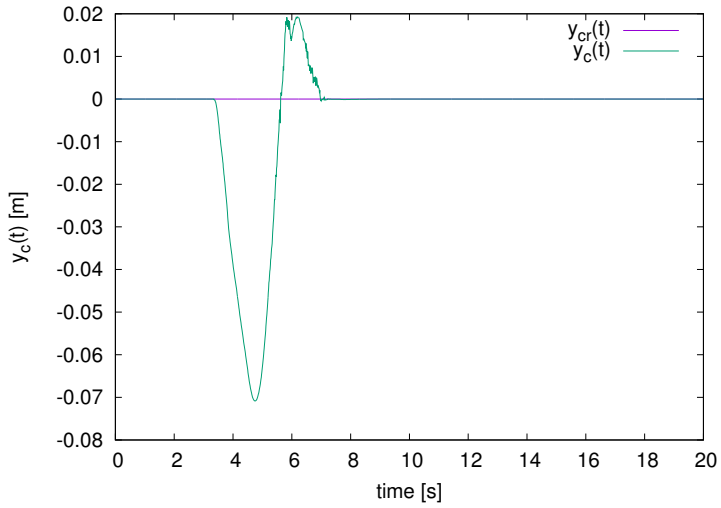

Fig. 14. Cartesian position $y_c(t) \times t$ for a step reference in X_0 direction.

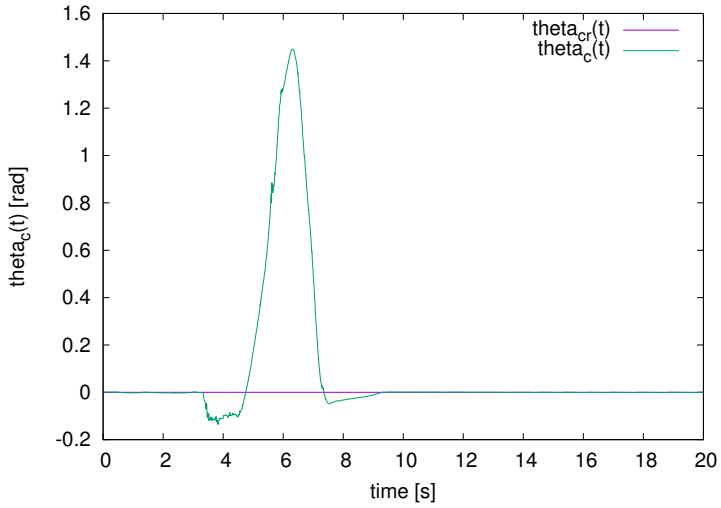

Fig. 15. Cartesian orientation $\theta_c(t) \times t$ for a step reference in X_0 direction.

The torque applied in the wheels to follow the trajectory is shown in Figure 16, while Figure 17 shows the intermediate variables $\boldsymbol{\eta}$ and $\boldsymbol{u}$. Note that components of $\boldsymbol{u}$ converge to the respective components of $\boldsymbol{\eta}$, as enforced by the backstepping procedure. Figure 18 shows the backstepping errors e_1 and e_2.

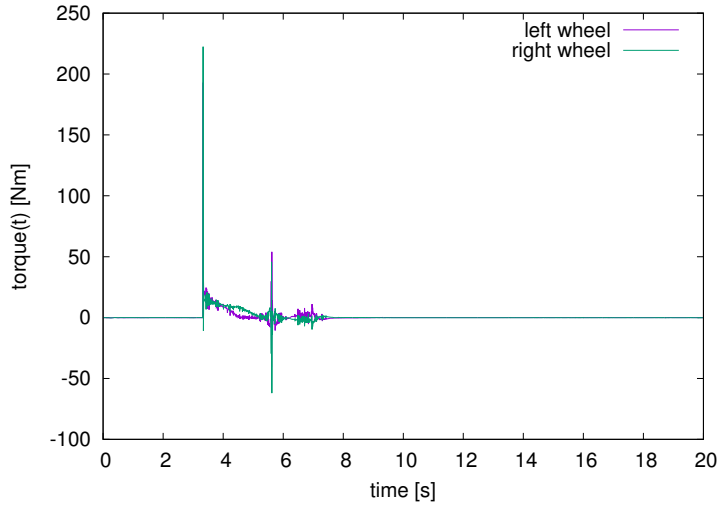

Fig. 16. Torque applied to wheels for a step reference in X_0 direction.

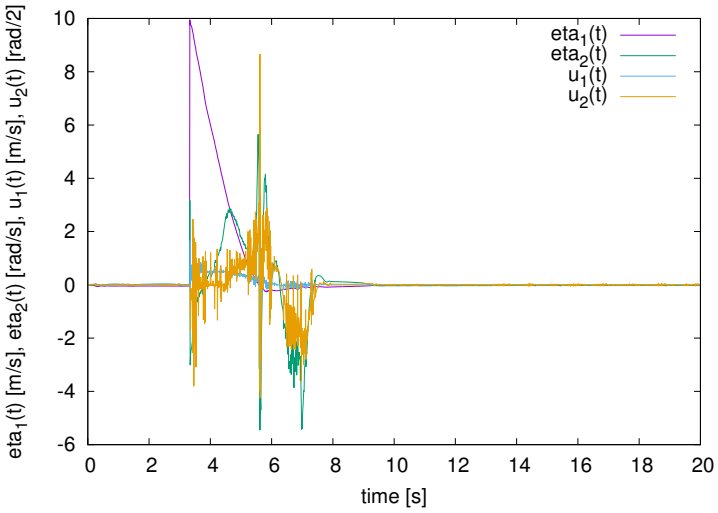

Fig. 17. Backstepping reference $\boldsymbol{\eta}(t)$ and $\boldsymbol{u}(t)$ for a step reference in X_0 direction.

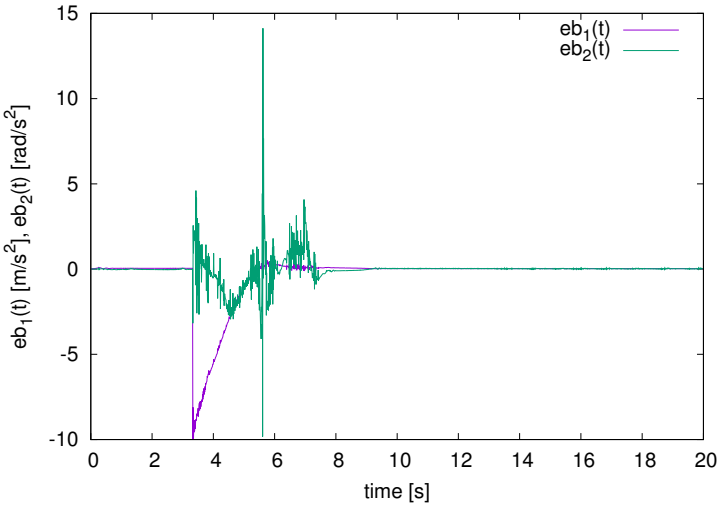

Fig. 18. Backstepping errors $e_1(t)$ and $e_2(t)$ for a step reference in X_0 direction.

Step in Y_0 Direction In this case, the reference is set to $x_c = 0$, $y_c = 1$ and $\theta = 0$, by issuing the command:

```
rosrun nonsmooth_backstep_controller pose_step.sh 0 1 0
```

Note that, as the robot can not move instantaneously in the Y_0 direction, this is a difficult motion for any controller.

Figure 19 shows the Cartesian position of the robot while following the step reference in Y_0 direction. In this figure it is possible to see the offset at the initial point of the trajectory and the robot convergence to the trajectory. Note that the tracking error is small and in part due to the odometry based pose estimation.

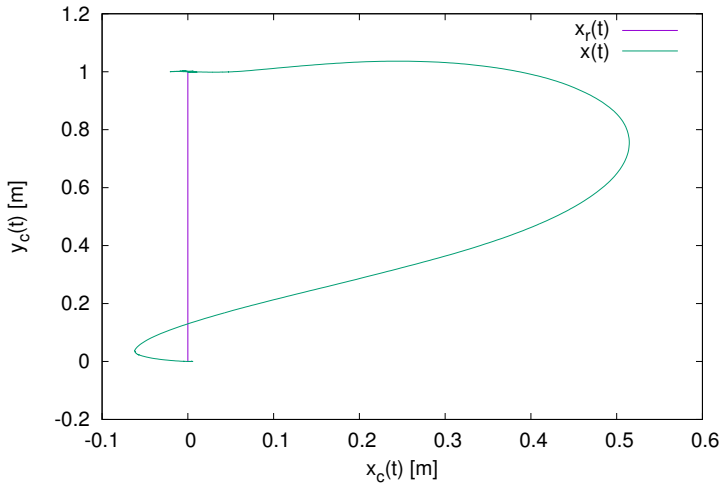

Fig. 19. Cartesian position $y_c(t) \times x_c(t)$ for a step reference in Y_0 direction.

Figures 20, 21 and 22 show the reference pose and the robot pose over the time. Those figures show the transitories of $x_c(t)$, $y_c(t)$ and $\theta_c(t)$ while they converge to the reference values.

The torque applied in the wheels to follow the trajectory is shown in Figure 23, while Figure 24 shows the intermediate variables $\boldsymbol{\eta}$ and $\boldsymbol{u}$. Note that components of $\boldsymbol{u}$ converge to the respective components of $\boldsymbol{\eta}$, as enforced by the backstepping procedure. Figure 25 shows the backstepping errors e_1 and e_2.

6.2 Trajectory Tracking

A launch file is provided to launch a test scenario where the nonsmooth backstepping controller is used to follow a trajectory in for of an 8. This launch files,

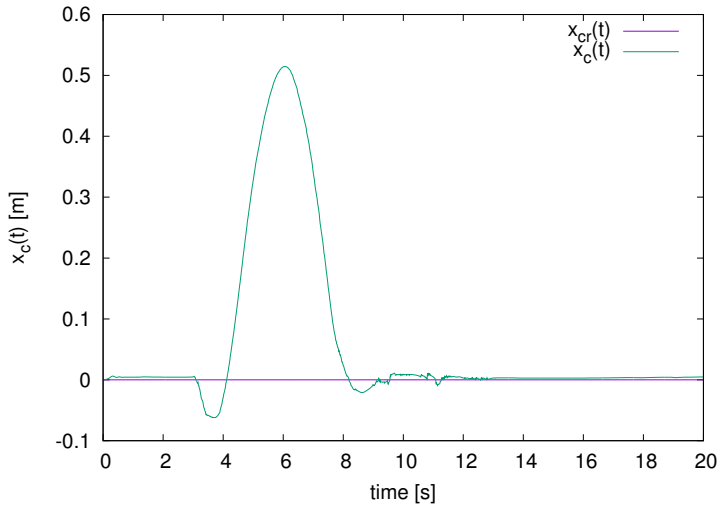

Fig. 20. Cartesian position $x_c(t) \times t$ for a step reference in Y_0 direction.

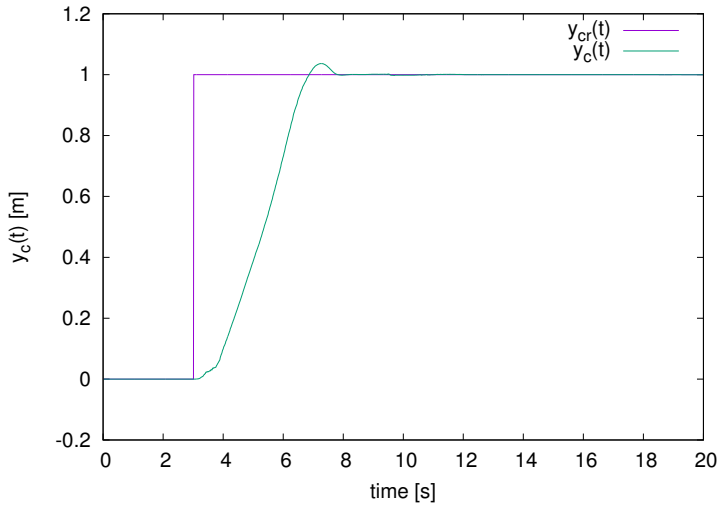

Fig. 21. Cartesian position $y_c(t) \times t$ for a step reference in Y_0 direction.

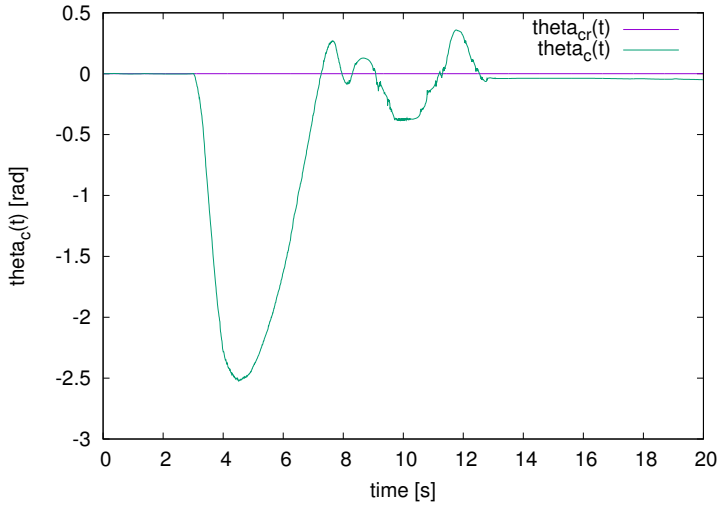

Fig. 22. Cartesian orientation $\theta_c(t) \times t$ for a step reference in Y_0 direction.

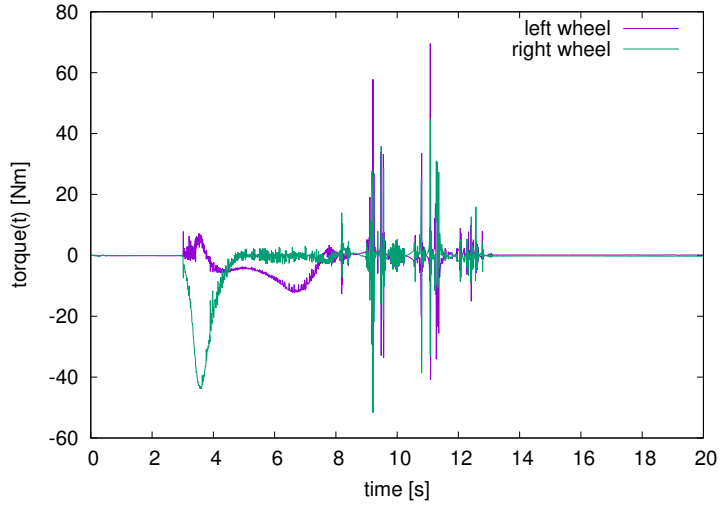

Fig. 23. Torque applied to wheels for a step reference in Y_0 direction.

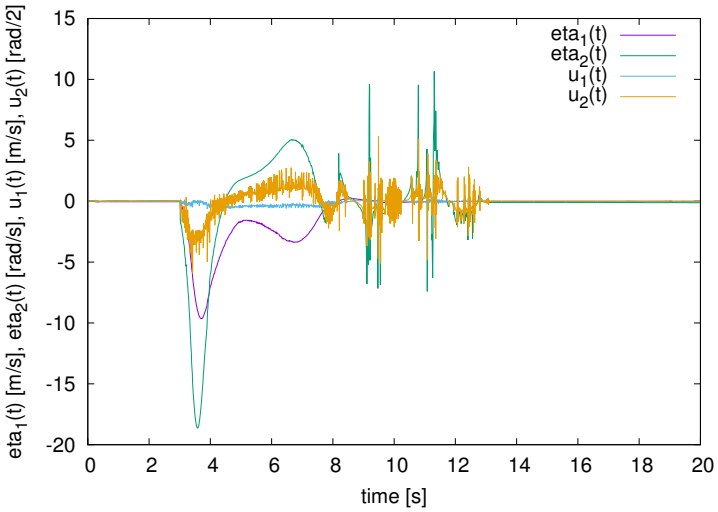

Fig. 24. Backstepping reference $\boldsymbol{\eta}(t)$ and $\boldsymbol{u}(t)$ for a step reference .

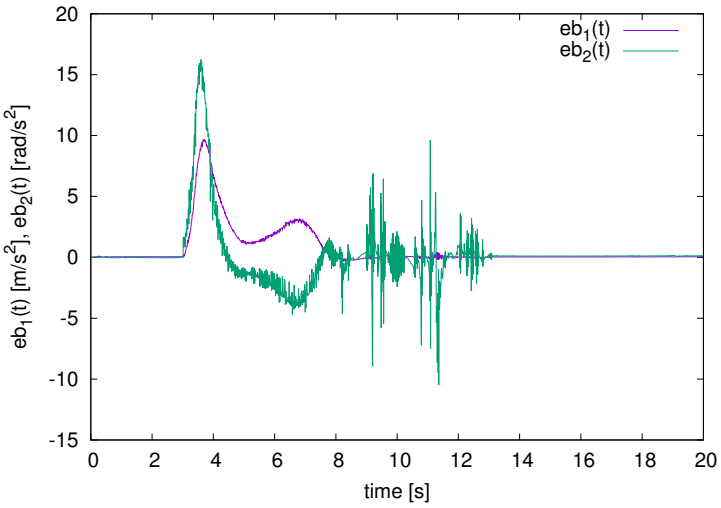

Fig. 25. Backstepping errors $e_1(t)$ and $e_2(t)$ for a step reference in Y_0 direction.

launches the backsteping controller, a tajectory generator, the Gazebo simulator and a visualization in Rviz. It can be launched with the command:

```
roslaunch nonsmooth_backstep_controller gazebo8.launch
```

.

Both, the Gazebo simulator and the Rviz visualization are launched because it is more convenient to see reference trajectory and the trajectory performed by the robot in Rviz, while the simulation in Gazebo is more realistic.

The Gazebo simulator is started in paused mode so that the initial condition for the simulation can be checked. By pressing the play button, the simulation starts.

Figure 26 shows this initial condition in Rviz. The blue arrow shows the reference trajectory point and the read arrow shows the current pose of the TWIl robot. Note that at the start of the simulation, the current pose of the robot $(0, 0, 0)$ and the initial pose of the reference trajectory $(0, -0.5, 0)$ are not the same. Therefore, the robot should converge to the trajectory and then follow it.

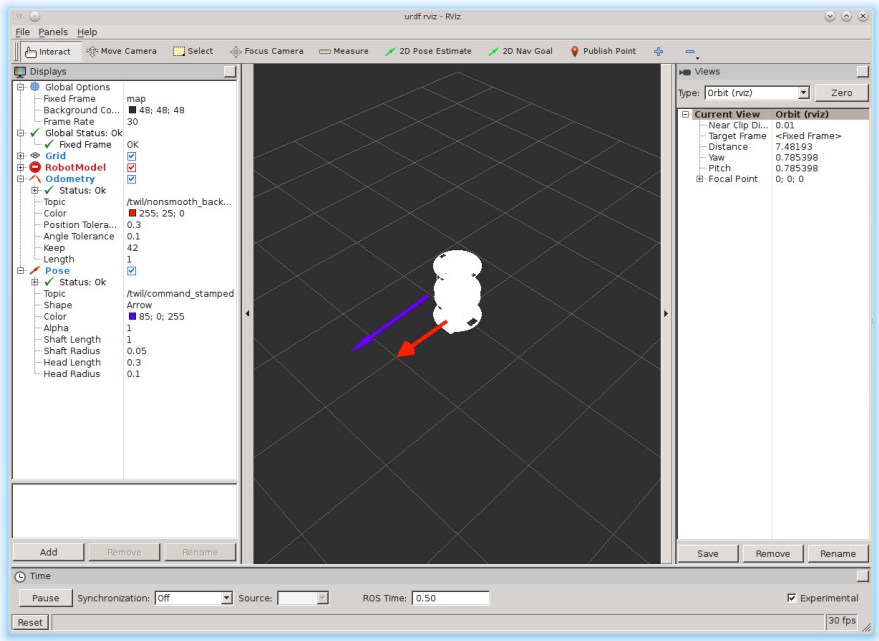

Fig. 26. Reference trajectory pose and Twil initial pose in Rviz.

Figure 27 shows the situation after 100 s. The read arrows show the history of poses of the TWIL robot, while the blue arrow shows the next pose to be followed. Note that in the start, the robot robot moves in order to converge to the trajectory and that is a difficult motion as this type of robot can not move sideways. The controller should then move away from the trajectory in order to be able to correct the offset in Y direction an then correct the orientation to track the trajectory. As those objectives are conflicting, the motion of the robot seems to be erratic, but it finally converge to the reference trajectory

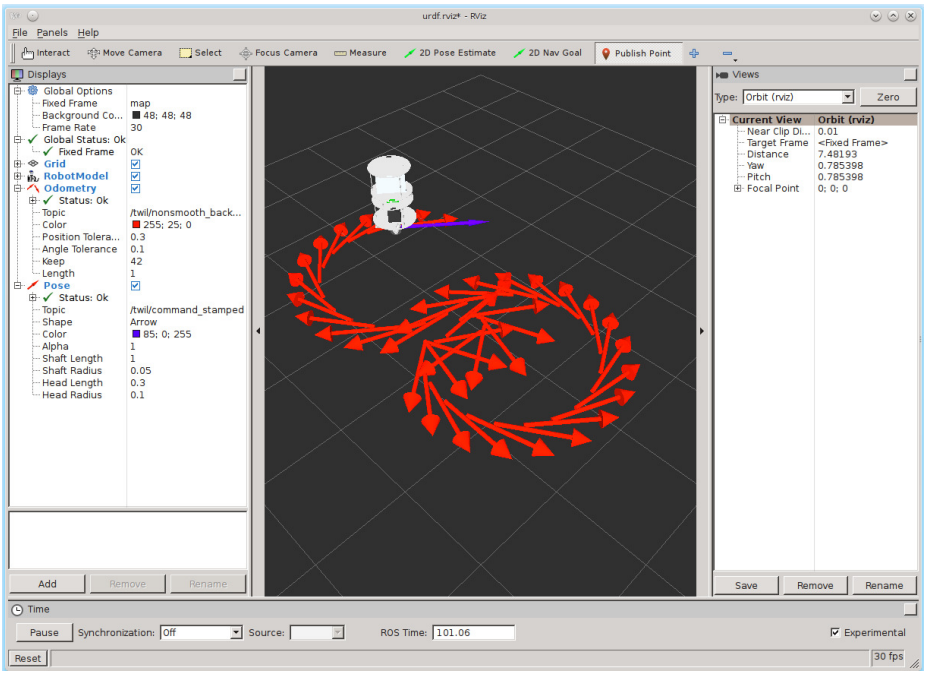

Fig. 27. Reference trajectory pose and Twil pose in Rviz after 100 s.

Figure 28 shows the computation graph for the node launched by the launch file. Note that the controller itself is not represented by a node since it is a plugin. The `/nonsmooth_backstep_controller/command` topic receives the reference and the `/tf` topic publishes the pose of the robot.

Figure 29 shows the Cartesian position of the robot while following the 8 reference trajectory. In this figure it is possible to see the offset at the initial point of the trajectory and the robot convergence to the trajectory. Note that the tracking error is small and in part due to the odometry based pose estimation.

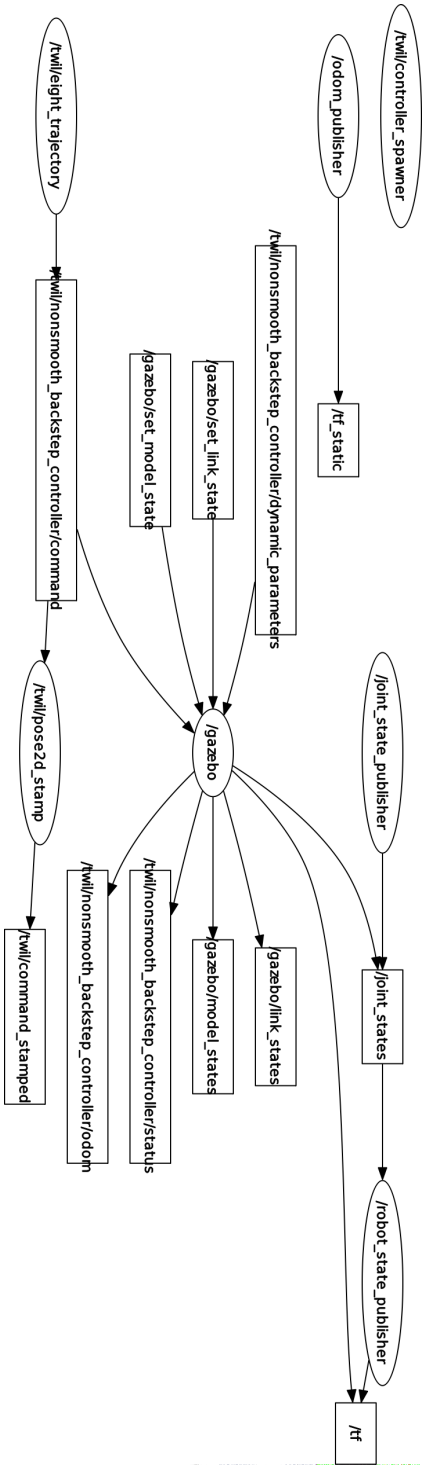

Fig. 28. Computation graph for the backstepping controller with trajectories displayed in Rviz.

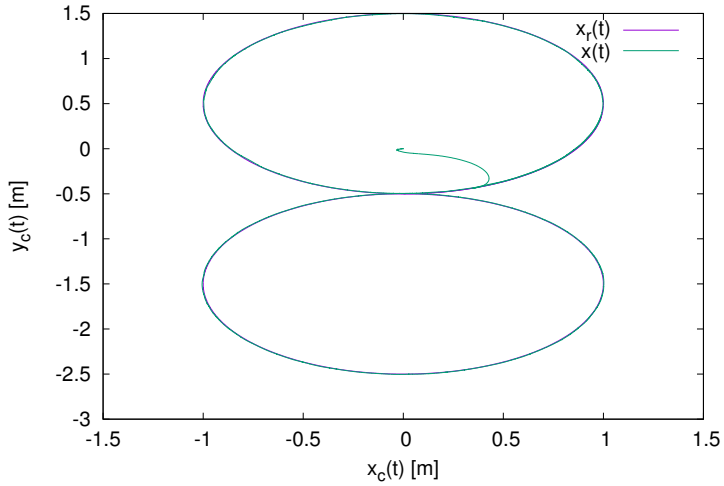

Fig. 29. Cartesian position $y_c(t) \times x_c(t)$ with the backstepping controller.

Figures 30, 31 and 32 show the reference pose and the robot pose over the time. It can be seen the initial transitory converging to the reference and then its following with a very small error.

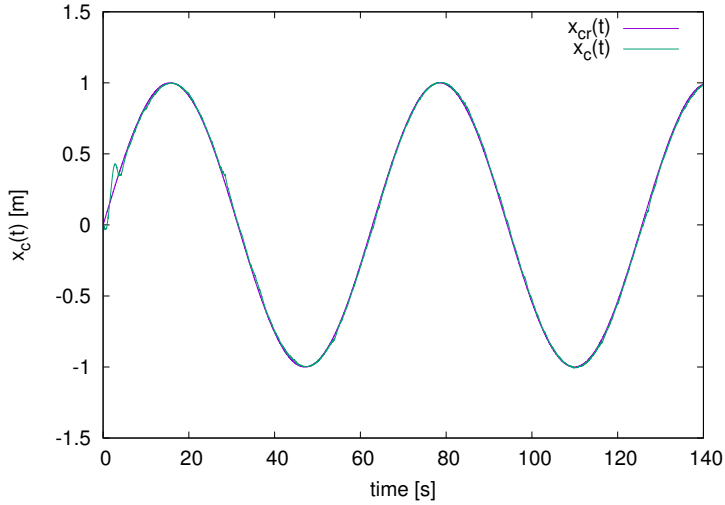

Fig. 30. Cartesian position $x_c(t) \times t$ with the backstepping controller.

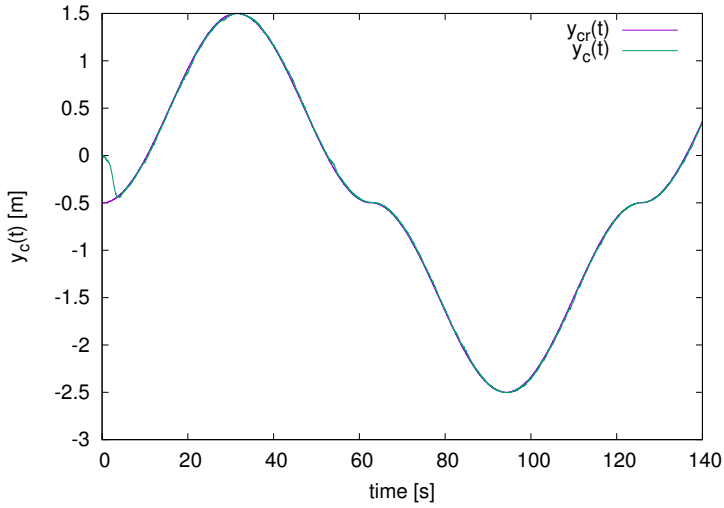

Fig. 31. Cartesian position $y_c(t) \times t$ with the backstepping controller.

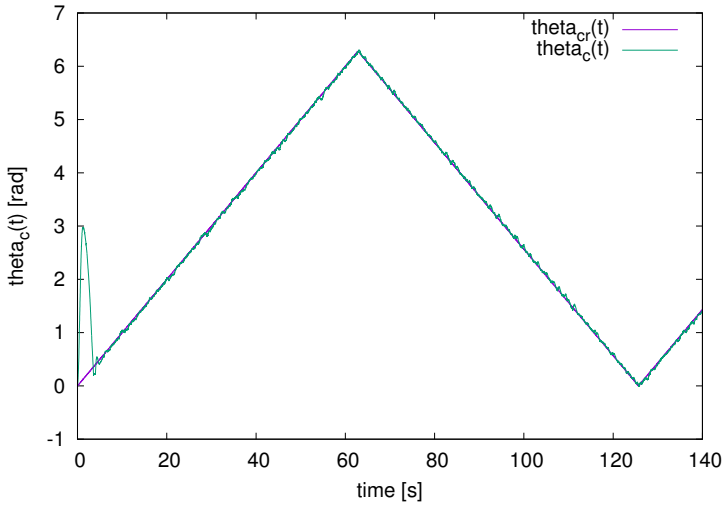

Fig. 32. Cartesian orientation $\theta_c(t) \times t$ with the backstepping controller.

The torque applied in the wheels to follow the trajectory is shown in Figure 33, while Figure 34 shows the intermediate variables $\boldsymbol{\eta}$ and $\boldsymbol{u}$. Note that com-

ponents of $\boldsymbol{u}$ converge to the respective components of $\boldsymbol{\eta}$, as enforced by the backstepping procedure. Figure 35 shows the backstepping errors e_1 and e_2.

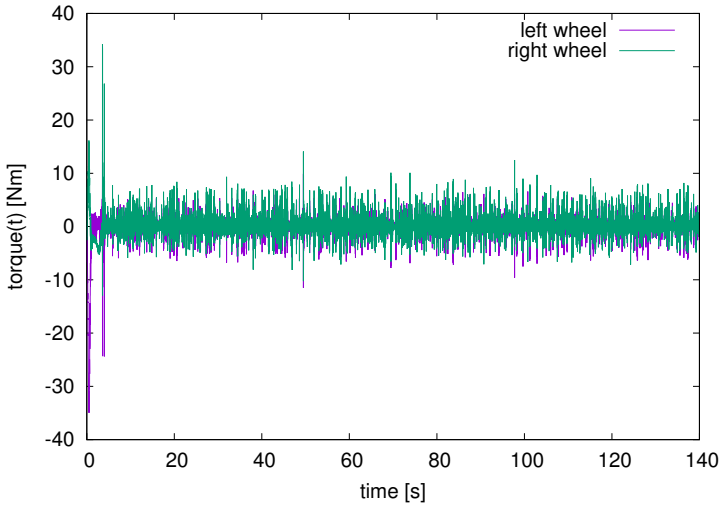

Fig. 33. Torque applied to wheels with the backstepping controller.

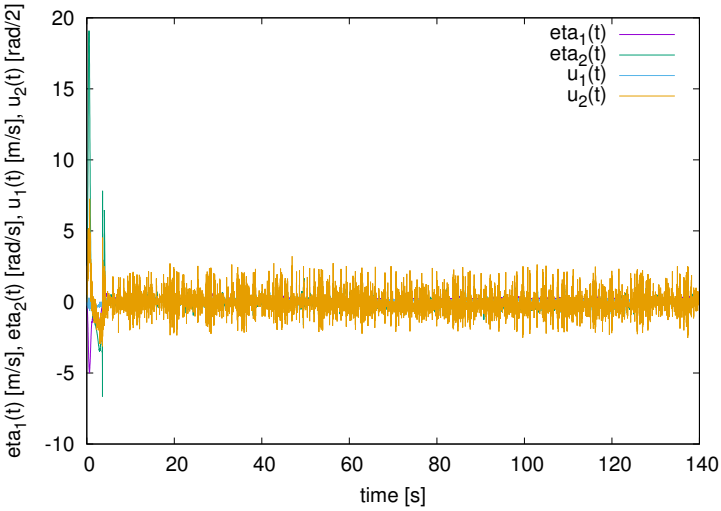

Fig. 34. Backstepping reference $\boldsymbol{\eta}(t)$ and $\boldsymbol{u}(t)$.

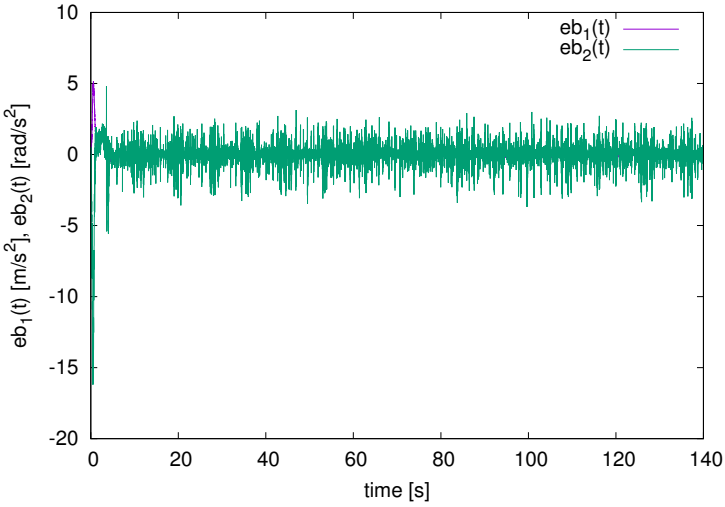

Fig. 35. Backstepping errors $e_1(t)$ and $e_2(t)$.

7 Conclusion

This work presented a controller for a mobile robot which considers the dynamic model of robot. This controller uses feedback linearization to compensate for the part of the mobile robot model related to its dynamics and then a non-smooth coordinate transform to enable the development of a Lyapunov-based non-linear control law. Then, backstepping is used to compute the appropriate inputs for the linearizing controller. The convergence of the system to the reference is proved by using the Barbalat lemma. The proposed controller was implemented in ROS as a MIMO non-linear controller, which is an important departure from the traditional low-level ROS controllers which consider a SISO system using PID controllers. The performance of the controller for the setting point and the trajectory tracking problems were shown using the Gazebo simulator and Rviz.

8 Authors Biographies

Walter Fetter Lages graduated in Electrical Engineering at Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) in 1989 and received the M.Sc. and D.Sc. degrees in Electronics and Computer Engineering from Instituto Tecnológico de Aeronáutica (ITA) in 1993 and 1998, respectively. From 1993 to 1997 he was an assistant professor at Universidade do Vale do Paraíba (UNIVAP), from 1997 to 1999 he was an adjoint professor at Fundação Universidade Federal do Rio Grande (FURG). In 2000 he moved to the Universidade Federal do Rio

Grande do Sul (UFRGS) where he is currently a full professor. In 2012/2013 he held a PostDoc position at Universität Hamburg. Dr. Lages is a member of IEEE, ACM, the Brazilian Automation Society (SBA) and the Brazilian Computer Society (SBC).

References

1. Aicardi, M., Casalino, G., Bicchi, A., Balestrino, A.: Closed loop steering of unicycle-like vehicles via lyapunov techniques. IEEE Robotics and Automation Magazine 2(1), 27–35 (1995)
2. Astolfi, A.: On the stabilization of nonholonomic systems. In: Proceedings of the 33rd IEEE American Conference on Decision and Control. pp. 3481–3486. Piscataway, NJ, IEEE Press, Lake Buena Vista, FL (Dez 1994)
3. Barros, T.T.T., Lages, W.F.: Development of a firefighting robot for educational competitions. In: Proceedings of the 3rd Intenational Conference on Robotics in Education. Prague, Czech Republic (2012)
4. Brockett, R.W.: New Directions in Applied Mathematics. Springer-Verlag, New York (1982)
5. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. Addison-Wesley, Reading, MA, third edn. (2001)
6. Campion, G., Bastin, G., D'Andréa-Novel, B.: Structural properties and classification of kinematic and dynamical models of wheeled mobile robots. IEEE Transactions on Robotics and Automation 12(1), 47–62 (Feb 1996)
7. Canudas de Wit, C., Sørdalen, O.J.: Exponential stabilization of mobile robots with nonholonomic constraints. IEEE Transactions on Automatic Control 37(11), 1791–1797 (Nov 1992)
8. Godhavn, J., Egeland, O.: A lyapunov approach to exponential stabilization of nonholonomic systems in power form. IEEE Transactions on Automatic Control 42(7), 1028–1032 (Jul 1997)
9. Goebel, P.: ROS by Example. Lulu, Raleigh, NC (Abr 2013), http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-hydro-volume-1/paperback/product-21460217.html
10. Isidori, A.: Nonlinear Control Systems. Springer–Verlag, Berlin, third edn. (1995)
11. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004). vol. 3, pp. 2149–2154. IEEE Press, Sendai, Japan (Sep 2004)
12. Kokotović, P.V.: Developments in nonholonomic control problems. IEEE Control Systems Magazine 12(3), 7–17 (Jun 1992)
13. Lages, W.F.: Implementation of real-time joint controllers. In: Koubaa, A. (ed.) Robot Operating System (ROS): The Complete Reference (Volume 1), Studies in Computational Intelligence, vol. 625, pp. 671–702. Springer International Publishing, Switzerland (2016)
14. Lages, W.F.: Parametric identification of the dynamics of mobile robots and its application to the tuning of controllers in ros. In: Koubaa, A. (ed.) Robot Operating System (ROS): The Complete Reference (Volume 2), pp. 191–229. Studies in Computational Intelligence, Springer International Publishing, Cham, Switzerland (2017), ⟨http://dx.doi.org/10.1007/978-3-319-54927-9_6⟩

15. Lages, W.F., Alves, J.A.V.: Differential-drive mobile robot control using a cloud of particles approach. International Journal of Advanced Robotic Systems 14(1) (2017), ⟨http://dx.doi.org/10.1177/1729881416680551⟩
16. Lages, W.F., Hemerly, E.M.: Smooth time-invariant control of wheeled mobile robots. In: Proceedings of The XIII International Conference on Systems Science. Technical University of Wrocław, Wrocław, Poland (1998)
17. Lucibello, P., Oriolo, G.: Robust stabilization via iterative state steering with an application to chained-form systems. Automatica 37(1), 71–79 (Jan 2001)
18. Maciel, E.H., Henriques, R.V.B., Lages, W.F.: Control of a biped robot using the robot operating system. In: Proceedings of the 6th Workshop on Applied Robotics and Automation. Sociedade Brasileira de Automática, São Carlos, SP, Brazil (2014)
19. Marder-Eppstein, E.: Navigation Stack (2016), available: ⟨http://wiki.ros.org/navigation⟩
20. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., Konolige, K.: The office marathon: Robust navigation in an indoor office environment. In: 2010 IEEE International Conference on Robotics and Automation (ICRA). pp. 300–307. IEEE Press, Anchorage, AK (May 2010)
21. Martinez, A., Fernández, E.: Learning ROS for Robotics Programming. Packt Publishing, Birmingham, UK (Sep 2013)
22. Murata, S., Hirose, T.: On board locating system using real–time image processing for a self–navigating vehicle. IEEE Transactions on Industrial Electronics 40(1), 145–153 (Feb 1993)
23. O'Kane, J.M.: A Gentle Introduction to ROS. CreateSpace Independent Publishing Platform (Oct 2013), available at http://www.cse.sc.edu/~jokane/agitr/
24. Pomet, J.B., Thuilot, B., Bastin, G., Campion, G.: A hybrid strategy for the feedback stabilization of nonholonomic mobile robots. In: Proceedings of the IEEE International Conference on Robotics and Automation. pp. 129–134. IEEE Press, Nice, France (Mai 1992)
25. Popov, V.M.: Hyperstability of Control Systems, Die Grundlehren der matematischen Wissenshaften, vol. 204. Springer-Verlag, Berlin (1973)
26. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. In: Proceedings of the IEEE International Conference on Robotics and Automation, Workshop on Open Source Robotics. IEEE Press, Kobe, Japan (May 2009)
27. Rehman, F., Rafiq, M., Raza, Q.: Time-varying stabilizing feedback control for a sub-class of nonholonomic systems. European Journal of Scientific Research 53(3), 346–358 (May 2011)
28. Slotine, J.J.E., Li, W.: Applied Nonlinear Control. Prentice–Hall, Englewood Cliffs, NJ (1991)
29. Sørdalen, O.J.: Feedback Control of Nonholonomic Mobile Robots. Thesis (dr. ing.), The Norwegian Institute of Technology, Trondheim, Norway (1993)
30. Sucan, I.A., Chitta, S.: MoveIt! (2015), [Online] Available: ⟨http://moveit.ros.org⟩
31. Teel, A.R., Murray, R.M., Walsh, G.C.: Non-holonomic control systems: from steering to stabilization with sinusoids. International Journal of Control 62(4), 849–870 (1995)
32. Zelenak, A., Pryor, M.: Stabilization of nonlinear systems by switched lyapunov function. In: Proceedings of the ASME 2015 Dynamic Systems and Control Conference. The Ammerican Society of Mechanical Engineers, Columbus, OH (2015), ⟨http://dx.doi.org/10.1115/DSCC2015-9650⟩