POLITECNICO DI TORINO
Computer Engineering

Master Degree Thesis

Data-driven Model and Trajectory Tracking SMC for a UGV system

Supervisors
Prof. Elisa Capello
Dr. Davide Carminati
Dr. Iris David Du Mutel

Candidate
Esmeraldi Xuna

July 2022

*To my family,
Ardian, Mimoza and
Romina*

Summary

One of the most relevant features for the design of control algorithms is to show their robustness. For this reason, the main objective of this thesis has been the design of a robust controller for the target Unmanned Ground Vehicle (UGV) by means of Sliding Mode Control (SMC) technique, a well-known control strategy that provides this desired feature. To reach such objective, a new firmware for the UGV has been designed and a data-driven model has been built.
The classical controller design procedure requires an initial characterization of a model which should be quite realistic but light, in order to make fast and reliable simulations. In this case, the provided model is a data-driven one, making possible the use of a kinematic model only of the UGV, keeping out dynamical modeling. A System Identification (SI) procedure has been necessary and has involved the collection of data, pre-processing, and finally model construction and validation; this phase confirmed the effectiveness of SI techniques and models, and was essential to refine the nonlinear auto regressive exogenous (NARX) model used for controller design and simulations purposes.
In the second phase of this thesis, a new firmware for the UGV has been implemented, introducing system states through a finite state machine (FSM); its structure is inspired from some well-known frameworks related to UAVs. The firmware provides also Ethernet communications with another board, where some ROS nodes run. Those ROS nodes are the designed controller (Trajectory Tracking SMC), the reference generator (Artificial Potential Fields technique, APF), and the navigation one (Extended Kalman Filter, EKF). An important result of such firmware structure is the modularity: it makes possible to test different control algorithms just by substituting the ROS node, but also the APF and EKF node could be substituted to test different navigation and reference generator algorithms. This modularity feature makes also easier the replacement of sensing and/or actuation components.
Finally, the new desired controller has been designed and tested in MATLAB/Simulink environment, where good performances have been observed in simulations, confirming the reaching of a robust control law (compared to the previous Proportional-Integral-Derivative controllers). Some tuning of the controller gains has been made and the block diagram of the controller subsystem has been translated into Python code manually, implementing a new ROS node.
At last, the goal was to test the physical effectiveness of the controller through laboratory experiments in different scenarios; results proved good performance in the proposed scenarios, where the time of target goal reaching has been reduced and the overall behavior of the UGV during experiments has been considered satisfying.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Unmanned Ground Vehicles (UGVs) are essentially a category of mobile robots, which can move autonomously to perform the desired task, and this last could be potentially everything spanning a wide range of opportunities for researchers, industries, and technical experts of various kind (e.g., mechanical, electronic, and computer engineers).
Extensive use of industrial manipulators has been already applied, but in the last years, companies are exploiting UGVs for logistics. Therefore, among the other applications of mobile robots, we can find handling military missions in a secure way and exploration of hazardous places. Related to the last one, we can find also space applications, like the Mars 2020 mission [15].
Every robot can be seen as a system, so a modeling phase of such a system is always required. This is the first of different transversal disciplines that we can find in robotics systems. Indeed, after system modeling, suitable control logic is necessary to actuate the robot to perform a task, opening wide opportunities for control engineers and also researchers in automatic controls field.

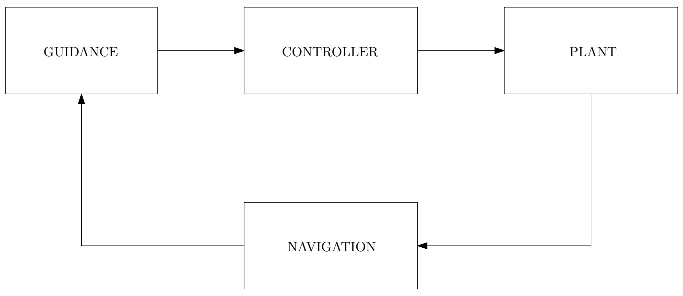


Figure 1.1: Control loop

The controller needs various kinds of inputs, and typically the logic behind it is the tracking of a reference signal satisfying some requirements. Hence, in order to track a signal, the knowledge of a system state is needed, leading to the use of navigation algorithms (i.e. algorithms able to estimate the configuration of the mobile robot). Instead, the "unmanned" side of a UGV is handled through guidance algorithms. To perform navigation

tasks, different sensors can be used depending on the application, while, for actuation purposes, actuators should be used. An example of the resulting structure of the so-called "control loop" implemented for UGVs is reported in figure 1.1.
After the overall design of the guidance, navigation, and control subsystems in a simulation environment, everything shall be deployed into a real system, which means microcontrollers placed in the chassis of the UGV. Generally speaking, such deployment requires an accurate firmware design in order to take advantage of the hardware resources provided by microcontrollers, since they are not as powerful as usual computers.
In this chapter, an overview of mobile robotics will be shown in the next section. Then:

- In **chapter 2** the kinematic model used for the target UGV and the system identification procedure applied will be shown, after a brief re-cap on some basic mathematical tools (i.e. reference frames and rotation matrices).
- In **chapter 3** will be presented the overall system architecture, and in particular the software one, since a new firmware has been implemented, as a re-adaptation of an already existing one developed for a drone.
- In **chapter 4** is reported the sliding mode control technique with a focus on the adopted law. After that, some simulation results are reported, including a comparison with the previous controllers designed for the UGV.
- In **chapter 5** some basic concepts of ROS are presented, and results of a simple mission without obstacles are described together with the results expected by means of simulation.

1.1 Mobile robotics overview

Unlike stationary ones (manipulators), mobile robots are equipped with a moving basis, leading to the possibility of motion in the working environment, which could be indoor or outdoor. In this thesis, an indoor application is planned, and basically, it consists of an autonomous motion toward a destination point, assuming known the initial configuration. Different types of mobile robots are available [13]:

- *land-based*: they can be also divided in different sub-categories, like the *wheeled* ones, *tracked* ones or *legged* ones. This thesis concerns the wheeled category, even if the UGV is a tracked one.
- *air-based*: also known as drones, or even unmanned aerial vehicles (UAVs).
- *wather-based.*

In [13] are shown advantages of wheeled mobile robots, mainly: easy to design, cheap, and there are no balancing problems of the structure. However, wheels present also disadvantages in environments that are not flat and where good friction is not available.
The target UGV has been considered equivalent to one equipped with two fixed standard wheels, allowing only the rotation over the contact point.
Another classification of mobile robots is based on drive systems [2]:

- *differential drive*: consists of a structure with two fixed wheels, each one autonomously actuated. Allows forward and backward motion, but also the rotation at the same point is allowed applying opposite commands to the wheels.
- *synchro drive*: all the fixed wheels are commanded by two inputs, one driving the orientation of the robot and the other one driving the longitudinal motion. It permits the same mobility of differential drive ones.
- *car-type*: two wheels are fixed and two are steerable, leading to the same structure of cars.

In figure 1.2 a basic representation of those 3 mentioned type of mobile robots is shown.

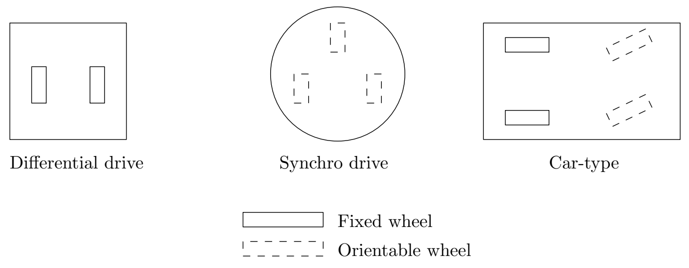


Figure 1.2: Mobile robots classification

As anticipated before, to operate autonomously appropriate sensors shall be placed into the chassis of the robot. These sensors can be classified in *proprioceptive/exteroceptive* and *passive-active*. Therefore, "it is important to characterize the sensor's performance using basic variables such as the dynamic range, power, resolution, linearity and bandwidth or frequency, sensitivity, error, accuracy, systematic errors, random errors, precision, and so on"[13]. Among the used sensors, we have [14]:

- *encoders*: they belong to the proprioceptive group, and are used to estimate angular velocities of wheels. Different technologies of optical encoders exists (e.g. incremental encoder, absolute encoder), but are also available based on Hall effect.
- *accelerometers*: used to measure accelerations along three axes. Measurements integration allows an estimate of velocities. Typically they are micro electro-mechanical systems (MEMS) and combined with other sensors (e.g. gyroscope) to form an inertial measurement unit (IMU).
- *gyroscope*: useful to measure angular velocities and orientation. They are based on a rotating or vibrating structure, but there are available kinds of gyroscope based on optical fibers, exploiting Sagnac effect.
- *magnetometer*: it is used to measure the magnetic field, exploited for heading evaluation of the robot.
- *RGB-D camera*: they provide coloured images featured with depth information, allowing further tasks (e.g. obstacle/object localization).

Therefore, the navigation task could include also a *mapping* task, which consists in determining a representation of the environment (i.e. a map) where the mobile robot operates. However, in this thesis, the map has been considered known and even the position of obstacles, when they are included in missions.
To perform the guidance task, a safe trajectory should be designed for the UGV. In [13], a classification of trajectory generation algorithms is available and the main groups are:

- *classical methods*: like cell decomposition and potential fields methods, characterized by high memory requirement and local minima drawbacks respectively.
- *probabilistic methods*: the so-called "probabilistic roadmap planner" belongs to this group.
- *heuristic planners*: known also as informed search strategies, the A* algorithm belongs to this group (it based on graph search theory).

Chapter 2

Mathematical model

In this chapter, the mathematical modelization of the target UGV will be presented, after a re-cap regarding some basic concepts about reference frames.
Typically, in standard control problem formulations, a set of kinematic and dynamic differential equations are exploited to formulate the control law. However, in this thesis project, only the kinematic equations were necessary, since the controller presented in chapter 4 involves only the kinematic model.
Therefore, the dynamical model has not been necessary since the plant used for controller development and simulations has been a black box one (i.e., a nonlinear auto-regressive with exogenous input model, NARX). Thus, after an illustration of System Identification concepts focused on the NARX model, the Devastator's model derivation is shown.

2.1 Basic concepts

In order to represent the position of the target UGV, a reference frame must be defined, or better to say, two: a local and a global one. One possible and common representation of reference frames is the Cartesian one, which has been used during simulations. The following concepts and notations about reference frames and rotation matrices are taken from [2].

To define such kind of reference frame, three unitary and mutually orthogonal vectors

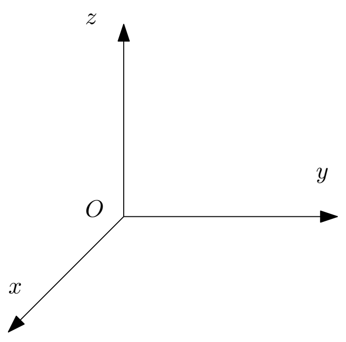


Figure 2.1: Reference frame $O - xyz$

shall be defined and it is common to denote them with $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$; they represent the x, y, and z-axis respectively of the *O-xyz* reference system, as shown in figure 2.1.
Once the reference frame is defined, any point can be represented with respect to it as a vector:

$$\boldsymbol{v} = v_x\boldsymbol{x} + v_y\boldsymbol{y} + v_z\boldsymbol{z}$$

which could be also denoted as:

$$\boldsymbol{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{2.1}$$

Given another reference frame $O' - x'y'z'$, with different orientation in $\mathbb{R}^3$, $\boldsymbol{x'}$, $\boldsymbol{y'}$, and $\boldsymbol{z'}$ can be derived as:

$$\begin{aligned} \boldsymbol{x'} &= x'_x\boldsymbol{x} + x'_y\boldsymbol{y} + x'_z\boldsymbol{z} \\ \boldsymbol{y'} &= y'_x\boldsymbol{x} + y'_y\boldsymbol{y} + y'_z\boldsymbol{z} \\ \boldsymbol{z'} &= z'_x\boldsymbol{x} + z'_y\boldsymbol{y} + z'_z\boldsymbol{z} \end{aligned}$$

and they represent the orientation of $\boldsymbol{x'}$, $\boldsymbol{y'}$, and $\boldsymbol{z'}$ with respect to $O - xyz$.
A more compact notation is the following one with rotation matrices:

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{x'} & \boldsymbol{y'} & \boldsymbol{z'} \end{bmatrix} = \begin{bmatrix} x'_x & y'_x & z'_x \\ x'_y & y'_y & z'_y \\ x'_z & y'_z & z'_z \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}'^T\boldsymbol{x} & \boldsymbol{y}'^T\boldsymbol{x} & \boldsymbol{z}'^T\boldsymbol{x} \\ \boldsymbol{x}'^T\boldsymbol{y} & \boldsymbol{y}'^T\boldsymbol{y} & \boldsymbol{z}'^T\boldsymbol{y} \\ \boldsymbol{x}'^T\boldsymbol{z} & \boldsymbol{y}'^T\boldsymbol{z} & \boldsymbol{z}'^T\boldsymbol{z} \end{bmatrix}$$

In this matrix, the elements are the direction cosines of the new reference frame's axes, i.e. *O'-x'y'z'* in this case. The column vectors of a rotation matrix represent a new reference frame, and some important features can be noticed:

- Mutual orthogonality between $\boldsymbol{x'}$, $\boldsymbol{y'}$, and $\boldsymbol{z'}$.
- Unitary magnitude of $\boldsymbol{x'}$, $\boldsymbol{y'}$, and $\boldsymbol{z'}$.

$\boldsymbol{R}$ is thus an orthonormal matrix, so the following two relationships hold:

$$\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}_3 \qquad \boldsymbol{R}^T = \boldsymbol{R}^{-1}$$

Another relevant feature of $\boldsymbol{R}$ is that $\det(\boldsymbol{R}) = 1$ in case of right-handed reference frame, while in case of left-handed holds $\det(\boldsymbol{R}) = -1$.
Given the structure of $\boldsymbol{R}$, three special matrices could be easily evaluated, which are the so-called elementary rotation matrices (i.e. the case where a rotation occurs with respect to a single axis). So, considering the previous case, the elementary rotation matrices of

$O' - x'y'z'$ with respect to $O - xyz$ are expressed as:

$$\boldsymbol{R}_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{R}_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix}$$

$$\boldsymbol{R}_x(\gamma) = \begin{bmatrix} 1 & 0) & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix}$$

Those elementary rotations give an important meaning to rotation matrices: they encode the angle of rotation around an axis in order to overlap the axis of $O - xyz$ into the ones of $O' - x'y'z'$, as it can be seen in figure 2.2.

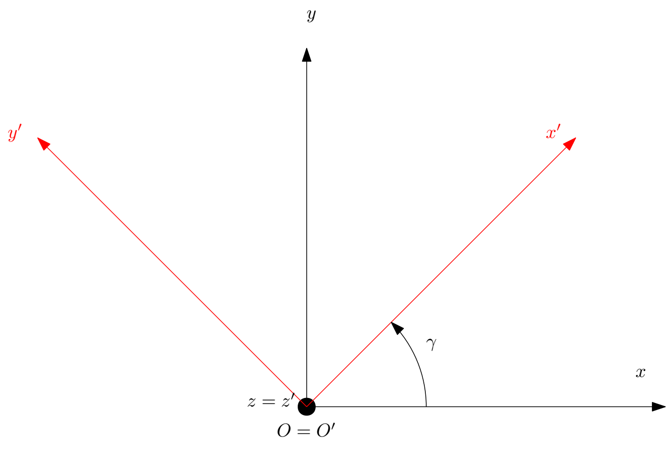


Figure 2.2: An elementary rotation around z axis

A generic vector in $O - xyz$ can be represent as stated in equation 2.1, but it can also be represented in $O' - x'y'z'$ as:

$$\boldsymbol{v'} = \begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix}$$

this means that the following relationship holds:

$$\boldsymbol{v} = v'_x\boldsymbol{x'} + v'_y\boldsymbol{y'} + v'_z\boldsymbol{z'} = \begin{bmatrix} \boldsymbol{x'} & \boldsymbol{y'} & \boldsymbol{z'} \end{bmatrix} \boldsymbol{v'}$$

and thus leading to:

$$\boldsymbol{v} = \boldsymbol{R}\boldsymbol{v'} \tag{2.2}$$

which is a really relevant relationship, since it let to immediately convert any vector from $O' - x'y'z'$ coordinates to $O - xyz$ coordinates.
Remembering that $\boldsymbol{R}^T = \boldsymbol{R}^{-1}$, we can also derive the inverse transformation from equation 2.2:

$$\boldsymbol{v'} = \boldsymbol{R}^T \boldsymbol{v}$$

About rotation matrices, the last key concept is the derivation of $\boldsymbol{R}$ from the ones representing elementary rotations. A rotation matrix is a redundant representation since the needed parameters to represent the orientation in $\mathbb{R}^3$ are only three: in literature can be found the Euler angles, which are an effective way to represent an orientation with only three needed parameters, and one example is the *roll-pitch-yaw* attitude representation, obtained with the following ordered steps:

- Rotation of ψ around x
- Rotation of θ around y
- Rotation of ϕ around z

Since the elementary rotations occurs with respect to a fixed frame, the overall rotation matrix will be obtained as:

$$\boldsymbol{R} = \boldsymbol{R}_z(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_x(\psi) \tag{2.3}$$

It is important to notice that in equation 2.3 the order of multiplication between elementary rotation matrices is not casual, since every elementary rotation is expressed with respect to initial fixed frames (in the case of representation with respect to moving axes the order changes, like in the *ZYZ* representation, which is not reported here).
Basic concepts reported above are the only necessary ones used in this thesis to represent UGV's position during simulations. More precisely, during simulations the fixed frame and the local frame will be coincident at start time: in particular, the x' axis will be oriented in the forward motion direction, the y' axis will be perpendicular to the x one and will be oriented on the left side with respect to the forward motion, and the z' axis will complete the right-handed reference frame. The reference frame $O' - x'y'z'$ will be the local one, while $O - xyz$ the fixed one, as in figure 2.3.
The next section contains the kinematic model and assumptions used to derive the objective control law.

2.2 Kinematic model

In the mobile robotics context, some well-known and quite realistic but approximated kinematic models are exploited for simulation purposes. The first model presented in Robotics courses to approach kinematic modeling is usually the unicycle model, and from this one, more complex robots can be modeled, like the two wheels differential drive one (i. e., the kind of mobile robot which is equivalent to the Devastator platform).
The following discussion about kinematic modeling considers the motion in a 2D plane only. Thus, about the topics presented in the previous section of this chapter, only rotations

around z-axis will be relevant. Again, the reference for topics and notations presented in this section is still [2].
The configuration of a mobile robot in a 2D plane can be represented by its position and orientation with respect to the fixed-reference through a vector $\boldsymbol{q}$, as illustrated in the following equation:

$$\boldsymbol{q} = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix}$$

The size of $\boldsymbol{q}$ will be denoted with n.
It is assumed that the position coordinates coincide with the center of mass of the UGV (i.e., the origin of the local reference frame), and it will be considered as a rigid body (some approximations are thus occurring). Figure 2.3 provides an example of a snapshot during a simulation: $O - xyz$ is the fixed reference frame, while $O' - x'y'z'$ is the moving one (i.e., local). So it is now clear the meaning of the vector $\boldsymbol{q}$: the first two coordinates indicates the position of the center of mass of the UGV, while the last one is the orientation of x' with respect to x (counter-clockwise).

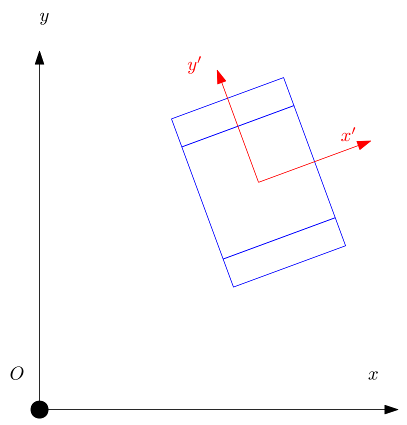


Figure 2.3: A configuration example during a simulation

Given a mobile robot, an important concept while threatening kinematic modeling is the accounting for nonholonomic constraints, which involves the generalized coordinates $\boldsymbol{q}$ and velocities $\dot{\boldsymbol{q}}$. Their general formulation is the following:

$$a_i(\boldsymbol{q}, \dot{\boldsymbol{q}}) = 0 \qquad i = 1, ..., k < n$$

nonholonomic constraints can be expressed in a linear way with respect to $\dot{\boldsymbol{q}}$:

$$\boldsymbol{a}_i^T(\boldsymbol{q})\dot{\boldsymbol{q}} = 0$$

then, considering all nonholonomic constraints together, we obtain:

$$\boldsymbol{A}^T(\boldsymbol{q})\dot{\boldsymbol{q}} = \boldsymbol{0}$$

A simple and classical example of nonholonomic constraint is the pure rolling motion of an ideal disk, which could be an initial approximated model of a wheel. This constraint is useful to keep out the lateral slippage from the model (i. e., $v_{y'} = 0$), and it is expressed by:

$$\dot{x}\sin\psi - \dot{y}\cos\psi = \begin{bmatrix}\sin\psi & -\cos\psi & 0\end{bmatrix}\dot{\boldsymbol{q}}$$

The possible trajectories which accounts for the nonholonomic constraints can then be derived solving the dynamical nonlinear system:

$$\dot{\boldsymbol{q}} = \sum_{j=1}^{m} \boldsymbol{g}_j(\boldsymbol{q})u_j = \boldsymbol{G}(\boldsymbol{q})\boldsymbol{u} \qquad m = n - k \tag{2.4}$$

Where $\boldsymbol{G}(\boldsymbol{q})$ is a basis (thus not unique) of the nullspace of matrix $\boldsymbol{A}^T(\boldsymbol{q})$. The vector $\boldsymbol{u} \in \boldsymbol{R}^m$ represents the inputs, and the equation 2.4 is the kinematic model of the mobile robot subject to the nonholomic constraints expressed in $\boldsymbol{A}(\boldsymbol{q})$.
For instance, in the case of a unicycle, the kinematic model obtained through solution of equation 2.4 could be:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\psi}\end{bmatrix} = \begin{bmatrix}\cos\psi\\ \sin\psi\\ 0\end{bmatrix} v + \begin{bmatrix}0\\ 0\\ 1\end{bmatrix} \omega \tag{2.5}$$

where v is the signed longitudinal velocity and ω the angular velocity of the wheel.
However, the target UGV is not an unicycle but a tracked mobile robot equivalent to a two wheeled differential drive one. In [4] it is shown that a two wheeled differential robot can be modeled as an unicycle applying the following relationships:

$$v = \frac{r(\omega_L + \omega_R)}{2} \tag{2.6}$$

$$\omega = \frac{r(\omega_R - \omega_L)}{B} \tag{2.7}$$

which are obtained taking into account nonholonomic constraints of pure rolling and no lateral slipping. In equations 2.7, r indicates the radius of wheels, B the distance between them, and $\omega_i, i = \{L, R\}$ the angular velocities of the two wheels.
The outputs of the plant used in simulations are the wheels angular velocities, so the kinematic model of the UGV is now fully available and exploitable.
To get the position and heading of the robot with respect to the fixed frame, a rotation matrix could be used and it is simply a rotation around z axis, in fact the following equation:

$$\begin{bmatrix}v_x\\ v_y\\ \dot{\psi}\end{bmatrix} = \begin{bmatrix}\cos(\psi) & -\sin(\psi) & 0\\ \sin(\psi) & \cos(\psi) & 0\\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}v_{x'}\\ v_{y'}\\ \dot{\psi}\end{bmatrix}$$

is equivalent to 2.5, since $v_{y'} = 0$ when the nonholonomic constraints are satisfied. Performing an integration, it is possible to derive the configuration of the target UGV expressed in the fixed reference frame (i.e. x and y coordinates plus the orientation ψ).

2.3 System identification

"System identification is a methodology for building mathematical models of dynamic systems using measurements of the input and output signals of the system"[3].
Deriving a model of a system can be done exploiting physical insights or collected experimental data, leading to two main approaches [6]:

- **First-principle modeling**: making some assumptions (leading to approximations) and applying fundamental principles of Physics, a set of equations modeling the system is then available. Obviously, these equations requires some physical parameters, which should be known a-priori or derived through experimental procedures; this is the critical point of such approach, it is not easy sometimes to estimate the mentioned parameters. In literature, this kind of model is named *white-box*.
- **System identification**: conversely from the previous approach, this one is named *black-box*. In this modeling technique, mathematical algorithms are applied in order to derive an input-output model from input-output data experimentally collected. The output of this procedure is not a set of equations, instead a set of some parameters (which, in general, have not physical meaning).
  Such approach is extremely powerful when the interest is only about an input-output model, however requires experimental data collection, processing of such data and finally the number of resulting parameters could be potentially large.
- **Mixed approach**: modeling is approached through Physics equations, whose parameters are then estimated with system identification procedures. It is a complex approach, and the resulting models are named as *gray-box*.

Since in classic control problem formulations, the first step has been the distinction of the system/model between linear time invariant and nonlinear ones. This initial step is still relevant in system identification modeling approach, because the model structure changes. In [7], four fundamental ingredients are reported:

- Observed data: they shall be unbiased and persistently exciting for the system. Therefore, some pre-processing before model constructions could be necessary.
- Candidate models: different models can be built from the already available and tested in literature. The choice of the model requires an initial classification of the system (i.e., linear or nonlinear).
- Fit criterion: this feature does not depend from the structure of the model selected. In general, given observed data and predicted one, an error is computed and the norm of such error should be minimized in order to have the best choice of parameters.
- Validation: fitting is measured on observed data, in order to measure the model accuracy.

In the following subsections, an example of linear model is provided. Then, the next subsection focuses on the NARX model, which has been the right choice to model the target UGV.

2.3.1 Least squares LTI system identification

Given a single-input single-output (SISO) system, and denoting by $u(t)$ the input at time t, and with $y(t)$ the output, a basic relationship between them is the linear difference equation [6]:

$$y(t) + a_1 y(t-1) + ... + a_n y(t-n) = b_1 u(t-1) + ... + b_m u(t-m), \quad m \leq n \tag{2.8}$$

Linear difference equations are implemented for discrete models, a choice which makes sense since the collected data are sampled ones. A-priori assumptions should be made on m and n, whose meaning is the number of previous inputs/outputs useful to determine the current prediction $y(t)$.
Equation 2.8 can be rewritten as:

$$y(t) = \hat{y}(t|\theta) = \phi^T(t)\theta$$

where ϕ is the vector of past input/output data and θ the vector of parameters:

$$\phi = [-y(t-1), .., -y(t-n), u(t-1), ..., u(t-m)]^T$$

$$\theta = [a_1, ..., a_n, b_1, ..., b_m]^T$$

From these vectors, mathematical algorithms could be exploited to determine parameters. The following parameters estimation approach showed is the Least Squares (LS) method [6].
For instance, given N observations, with N compatible with n and m such that the following system of linear equations is solvable through inversion of A:

$$y = A\theta$$

where A is constructed by rows defined with ϕ at different time instants:

$$\begin{bmatrix} y(n) & ... & y(1) & u(m+1) & ... & u(1) \\ y(n+1) & ... & y(2) & u(m+2) & ... & u(2) \\ ... & ... & ... & ... & ... & ... \\ y(N-1) & ... & y(N-n) & u(N) & ... & u(N-m) \end{bmatrix}$$

So, for A compatible with θ, the parameters are simply estimated as:

$$\hat{\theta} = A^{-1}y$$

where $\hat{\theta}$ indicates the estimation of the true parameters θ.
However, this is not a realistic case, since only N measures where provided. Due to the noise always present in measurement procedures, N should be much more higher than the number of parameters, leading to a "thin" matrix A.
So, taking enough samples to neglect measurement noise, e.g. with $N >> 2n+1$, the parameters are estimated through the pseudo-inverse of A (it is not anymore a square matrix):

$$\hat{\theta} = (A^T A^{-1})A^T y$$

Under some assumption, the so called *consistency property* holds, satisfying $\lim_{N\to\infty} \hat{\theta} = \theta$.
However, this subsection was necessary only to make an introduction on LTI system identification showing briefly the LS method, but in this thesis any LTI model have been identified, since they were not effective for the UGV.

2.3.2 NARX

About nonlinear system identification, a focus is made in this subsection about nonlinear autoregressive with exogeneous input models (NARX), since has been the effective model to get a data-driven plant. NARX is the nonlinear version of autoregressive with exogeneous input models (ARX).
An ARX(n_a, n_b) model with delay n_k is described by the difference equation [7]:

$$y(t) + a_1 y(t-1) + ... + a_n y(t-n_a) = b_1 u(t-n_k) + ... + b_m u(t-n_k-n_b) + e(t)$$

then, exploiting backward shift operator $q^{-1}x(t) = x(t-1)$, is then trivial to prove that the noise filter has the same poles of the system:

$$y(t) = \frac{B(q^{-1})}{A(q^{-1})}u(t) + \frac{1}{A(q^{-1})}e(t)$$

leading to the block diagram in figure 2.4.

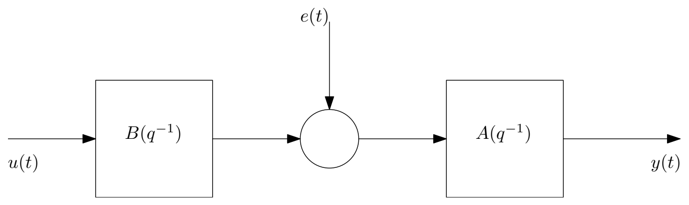


Figure 2.4: NARX model block diagram

However, when a linear model provides scarce fitting due to embedded nonlinearities of the system, a NARX structure should be exploited.
A SISO NARX model is described by:

$$y(t) = F(y(t-1), ..., y(t-n_a), u(t-n_k), ..., u(t-n_k-n_b)) + e(t)$$

In equation 2.3.2 can be noticed that a function F has been introduced: it is used to map the regressors in order to fit the output data. This is the key difference between ARX and NARX: in the latter a nonlinear map occurs in order to model nonlinear systems. This concept is the same on which foundations of neural networks have been built.
Different nonlinear function F structures are available, e.g., wavelet expansions, and sigmoidnet.
However, since the built model of the UGV has been a multiple-input multiple-output (MIMO) one, the difference equations of a MIMO NARX with m outputs and r inputs can be written as [4]:

$$\boldsymbol{y}(t) = \boldsymbol{F}(\boldsymbol{y}(t), \boldsymbol{u}(t), \boldsymbol{e}(t)) + \boldsymbol{e}(t)$$

with:

$$\boldsymbol{y}(t) = [y_1(t) \; ... \; y_m(t)]^T$$
$$\boldsymbol{u}(t) = [u_1(t) \; ... \; u_r(t)]^T$$
$$\boldsymbol{e}(t) = [e_1(t) \; ... \; e_m(t)]^T$$

so $\boldsymbol{F}(\cdot)$ is a matrix of nonlinear mappings.
Concluding, the predictions of the NARX model will then be computed as:

$$\hat{y}(t|\theta) = f(\phi(t), \theta)$$

where $f(\cdot)$ is the nonlinear function that maps the regressors to the output.

2.4 Target UGV model derivation

In order to derive a data-driven plant model some data is needed, and following the approach presented in [8, 4], they are the input pulse-width modulation signals (PWMs) applied to the motors and the output angular velocities of the wheels (i.e., the driving wheels of tracks).
Practically, the already existing firmware has been used to apply constant PWMs and the input applied together with the output of encoders have been registered through PuTTY into a CSV file. Such file has been then processed through MATLAB script to get the vectors of input/output data, as required by the MATLAB System Identification Toolbox (used for model construction). However, this approach already used in [4] was not sufficient to capture well the dynamics of the system, so those constant-input data were integrated with new samples where the input has been generated through joystick. In fact, applying joystick command and trying random commands was useful to refine the already existing model, obtaining a more precise one.
In figure 2.6 is reported an example of input data collected keeping constant PWMs, while in figure 2.5 the respective output data applying the shown input.

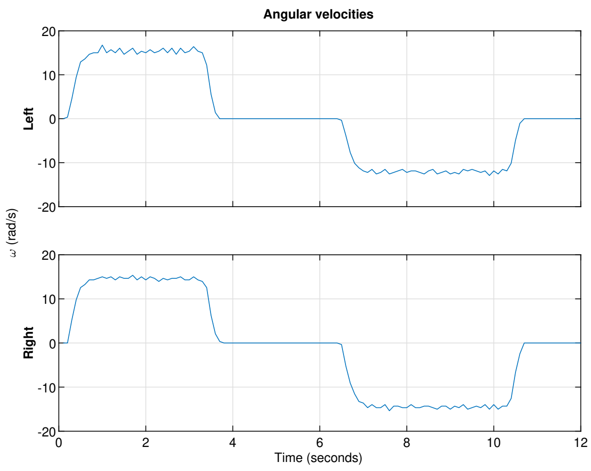


Figure 2.5: Constant PWMs output example

Similarly, in figure 2.8 an example of input data collected with random joystick command activity is reported, and in figure 2.7 the respective collected output.

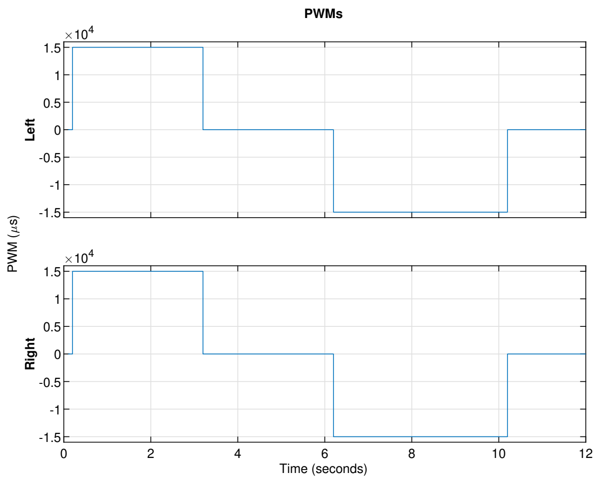


Figure 2.6: Constant PWMs input example

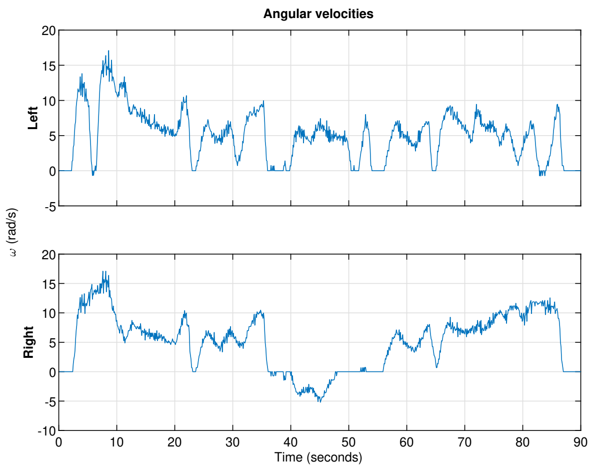


Figure 2.7: Joystick command output example

To build the model, all experiments through joystick and constant PWMs where concatenated into single vectors, preparing them for model building through the toolbox mentioned above. So, in figure 2.9 is shown the overall output data used for model construction, and in figure 2.10 the overall input one.

After this data collection phase, the choice of a model structure was the next step. Linear ones were already excluded in [4], leading to the conclusion that the MIMO NARX already choosed was the most suitable one for the UGV. Additionally, an attempt to decouple the model into two single-input single-output (SISO) NARX (i.e., one for each wheel)has been made but fitting was poor already in the training set, so this modeling approach has been

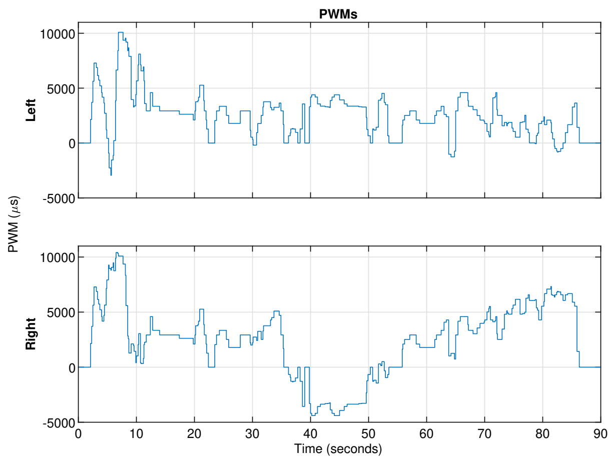


Figure 2.8: Joystick command input example

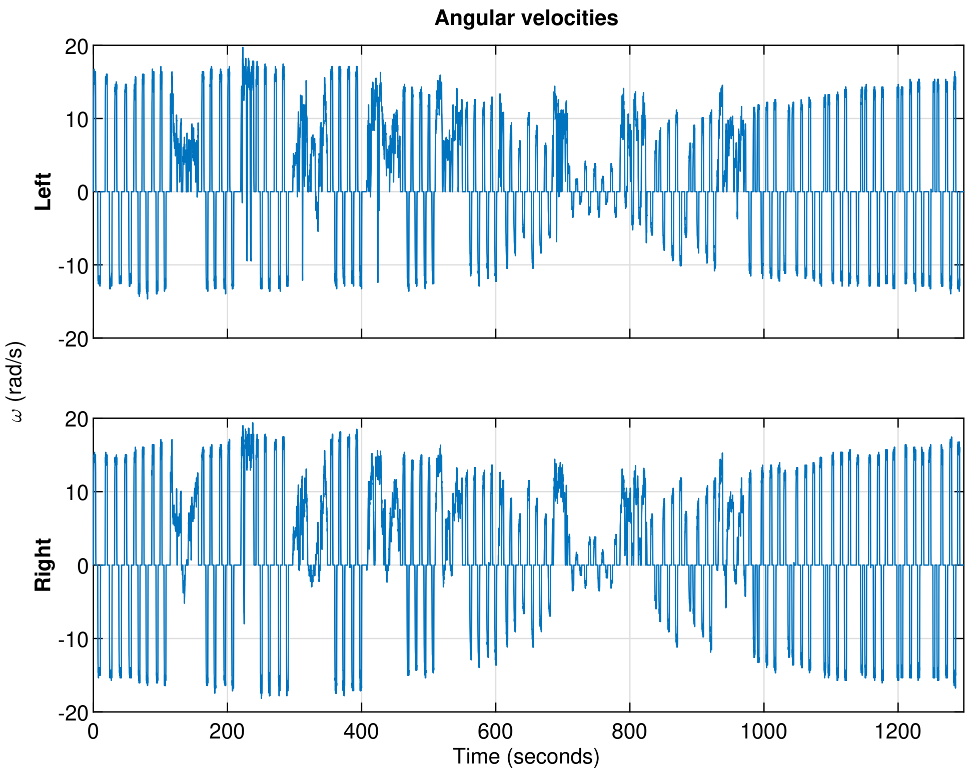


Figure 2.9: Concatenation of all examples outputs

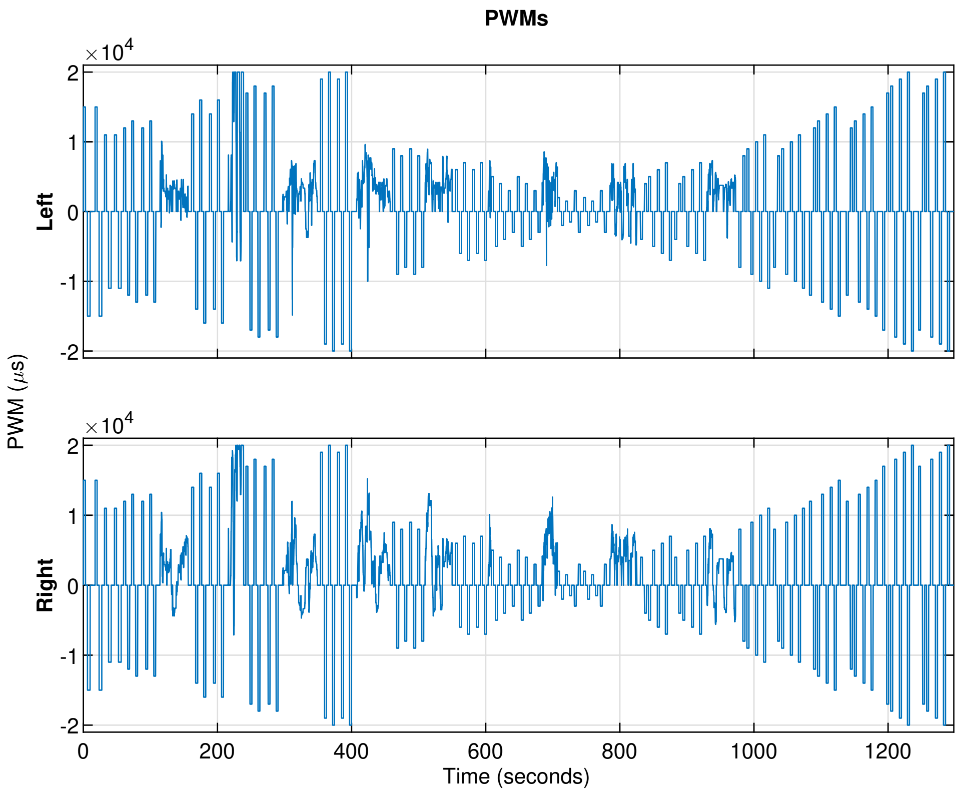


Figure 2.10: Concatenation of all examples inputs

also excluded.
The resulting MIMO NARX (generated through *nlarx* MATLAB command) is characterized by the orders:

$$n_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$n_b = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

$$n_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and the nonlinear mappings are two wavenets (77 units for the left wheel and 87 units for the right one). The orders n_a,n_b and n_c where tuned by trial and error to best fit the data, while for the nonlinear mappings, the wavenet chosen automatically by MATLAB was the most suitable one.
Therefore, the resulting FitPercent and Final Prediction Error were $[90.09; 90.87]\%$ and 0.304 respectively.
Finally, the model has been tested on some new data to test its effectiveness, which is a smaller dataset but with the same features of the training one (i.e., constant PWMs data

and joystick data). In figure 2.11 are reported the resulting fit of the generated model on some new samples (i.e., not seen at model construction phase).

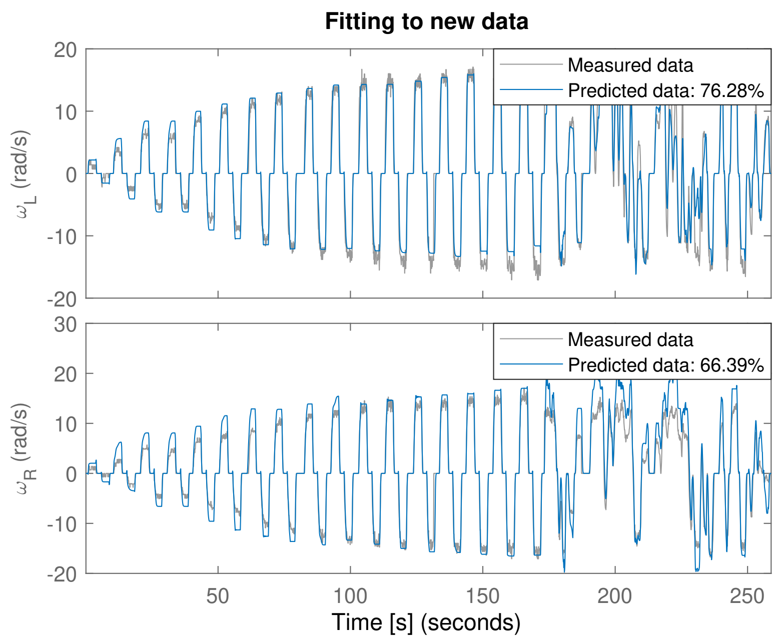


Figure 2.11: New samples fit

Compared to the already existing model developed in [4], the percentage fitting seems lesser. It is necessary to remark that this is due to the new joystick samples. In fact, if we consider the experiments with constant PWMs the two models have similar fitting on the new data, while considering joystick data, the new model has better performances since it is trained with more exciting inputs for the dynamics of the system.
Another relevant observation can be done about the relative fitting between the two wheels: the right one is less precise, and this behavior is consistent with the observed one during data collection, as a matter of fact, applying equal constant PWMs, the right track moved always slower than the left one, showing an asymmetric dynamic. For this reason, even if the results on the wheels are different, they have been accepted. Therefore, this proves the efficacy of NARX modeling: it caught well the actual behavior of the UGV.

Chapter 3

System architecture

Every robot is a multidisciplinary system since it involves four main areas:

- Mechanics.
- Electronics.
- Software.
- Control.

Mechanical modeling and electronic component selection were not dealt with in this thesis. However, in the next section, some information about the electronics (i.e., hardware) components mounted on the UGV will be shown, given their significance for guidance, navigation, and control (GNC) purposes.
Regarding the mechanical structure, figure 3.1 shows the target UGV within a detailed mechanical vision.

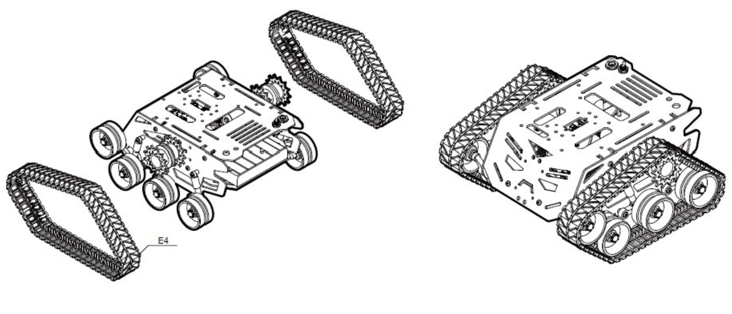


Figure 3.1: Devastator's structure [9]

As anticipated in chapter 2, the Devastator is a tracked mobile robot, but it is analogous to a two-wheeled differential drive one. The DC brushed motors actuate on one driving wheel per side, assimilating thus the behavior to a more straightforward mobile robot. So every track is actuated by one wheel, and the other four are used to support tracks (as

shown in figure 3.1).
As mentioned above, the actuation mechanism is handled through DC brushed motors commanded through pulse-width modulation (PWM) signals, which produce the desired voltage. PWMs are characterized by a signal duration *on* (i.e., pulse width), and by a period of signal duration (whose inverse corresponds to the frequency). The period of PWMs in this thesis was set to 20000µs, as already done in [4].
DC brushed motor converts the input electric energy into output mechanical energy, rotating the wheel. This kind of DC motor is based on the Lorentz law $\boldsymbol{F} = i\boldsymbol{L} \times \boldsymbol{B}$, where $\boldsymbol{L}$ is a vector with the wire length, i the current flowing and $\boldsymbol{B}$ the magnetic field vector. This is just a simplified model with one coil passing through a space between two permanent magnets with opposite polarization. Every DC motor is then characterized by an electromagnetic dynamic equation and a mechanical one, but in this thesis those relationships were not exploited to control the UGV.
Follows here a section of the hardware used to do physical experiments, while in section 3.2 will be shown the software.

3.1 Hardware level

The most clear representation of the hardware architecture is through the block diagram in figure 3.2

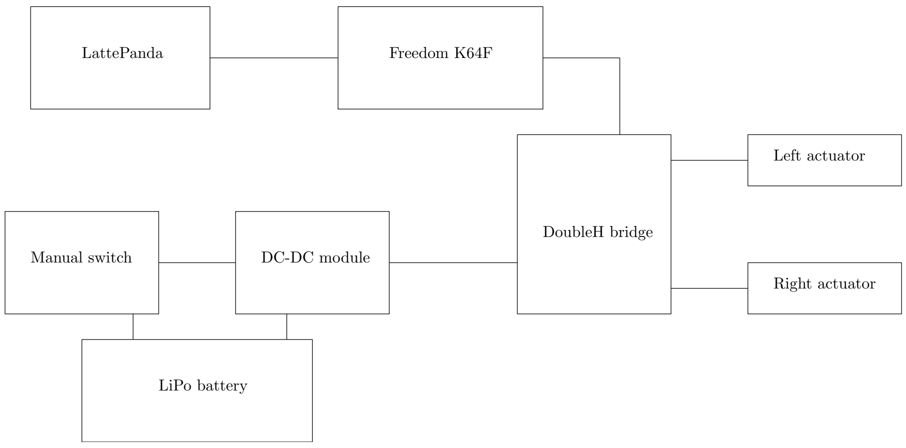


Figure 3.2: Block diagram of the hardware architecture

In more details:

- *LiPo battery*: this module is the power supply, connected by wiring to a manual switch and a DC-DC module; for security reasons, it is possible to switch off the power supply thanks to the manual switch.
- *Manual switch*: permits to stop the actuation at a hardware level in order to deal with situations where physical experiments fail.
- *DC-DC module*: handles the correct voltage supply for the double H bridge module.

- *Double H bridge*: the solution for motor driving. A double H bridge allows to control both motors independently.
- *Left/Right actuator*: DC brushed motors mentioned above.
- *LattePanda*: a computer whit Ubuntu installed. It runs some ROS nodes and communicates through Ethernet with the Freedom K64F for sensing and actuation data exchange.
- *Freedom K64F*: this microcontroller performs sensing and actuation with the commands received from LattePanda. Among the sensors, we also have the encoders (the other ones are indicated in the firmware section).

A RealSense depth detection camera is also available, and it is used by LattePanda through ROS. Therefore, another external IMU is connected to the Freedom K64F, making redundancy possible for accelerometers.
However, the hardware set-up was already given, and any kind of hardware architecture design has been performed in this thesis. A more interesting focus is made in the next section at software level, where more interesting details are given about GNC, while in the last section of the current chapter, the new firmware architecture designed and coded for the Freedom K64F will be shown.

3.2 Software level

In order to make simulations and physical experiments, some GNC details shall be presented. Will be provided the GNC algorithms used in Simulink and in ROS, during simulations and laboratory experiments respectively.

3.2.1 Guidance

This software module is responsible for path planning. In the scenarios considered, the space where the motion occurs is well known: it could be without obstacles or with obstacles with a known position (encoded into the guidance algorithm). The position of the UGV is known at every simulation step (through odometry or extended Kalman filter, as will be shown in the Navigation subsection).
A simple way to provide a guidance algorithm could be an a-priori known trajectory that shall be followed (i.e., trajectory tracking) during simulation steps. However, this approach always requires the set-up of the motion trajectory, so it is not so comfortable. A more interesting guidance algorithm is the artificial potential fields one, which provides the feature of autonomous motion planning.
The three main components in this algorithm are:

- Robot position: it must be determined at every simulation step (i.e. estimated through navigation algorithms).
- Goal position: shall be set when starting missions.
- Obstacle position: shall be set when starting missions.

The motion of the robot is guided by the influence of an artificial potential field U, which is evaluated as a sum of two ones, an attractive one and a repulsive one:

$$U(\boldsymbol{q}) = U_{att}(\boldsymbol{q}) + U_{rep}(\boldsymbol{q})$$

Mathematically, path planning through U means to set its global minimum to the goal point and to reach such minimum from the starting configuration. A method to reach this objective is the gradient descent one:

$$F(\boldsymbol{q}) = -\nabla U(\boldsymbol{q}) = -\nabla U_{att}(\boldsymbol{q}) - \nabla U_{rep}(\boldsymbol{q})$$

Gradient descent algorithm has a side effect: the potential fields shall be carefully chosen in order to have an unique global minima (i.e., this algorithm could suffer global minima effects, leading to stationary points).
The APF configuration is the same of [4], so the attractive field and the resulting force are defined as:

$$U_{att}(\boldsymbol{q}) = \begin{cases} \frac{1}{2}K\rho_f(\boldsymbol{q})^2 & \rho_f(\boldsymbol{q})^2 \leq d \\ dK\rho_f(\boldsymbol{q}) - \frac{1}{2}Kd^2 & \rho_f(\boldsymbol{q})^2 > d \end{cases}$$

$$\boldsymbol{F}_{att}(\boldsymbol{q}) = \begin{cases} -K(\boldsymbol{q} - \boldsymbol{q_g}) & \rho_f(\boldsymbol{q})^2 \leq d \\ \frac{-dK(\boldsymbol{q}-\boldsymbol{q_g})}{\rho_f(\boldsymbol{q})} & \rho_f(\boldsymbol{q})^2 > d \end{cases}$$

where K is a parameter for field intensity, $\boldsymbol{q}$ the robot position, $\boldsymbol{q_g}$ the goal position, $\rho_f(\boldsymbol{q})$ the distance between the actual position and the goal, and d a distance useful to change field in order to have a suitable attraction to the goal.
The repulsive field and force are instead defined as:

$$U_{rep}(\boldsymbol{q}) = \begin{cases} \frac{1}{2}\eta(\frac{1}{\rho(\boldsymbol{q})} - \frac{1}{\rho_0})^2 & \rho_f(\boldsymbol{q})^2 \leq \rho_0 \\ 0 & \rho_f(\boldsymbol{q})^2 > \rho_0 \end{cases}$$

$$\boldsymbol{F}_{rep}(\boldsymbol{q}) = \begin{cases} \eta(\frac{1}{\rho(\boldsymbol{q})} - \frac{1}{\rho_0})\frac{1}{\rho(\boldsymbol{q})^2}\nabla\rho(\boldsymbol{q}) & \rho_f(\boldsymbol{q})^2 \leq \rho_0 \\ 0 & \rho_f(\boldsymbol{q})^2 > \rho_0 \end{cases}$$

where ρ_0 is the influence area of each obstacle, $\rho(\boldsymbol{q})$ the shortest distance from the obstacle space and η a gain used to regulate the magnitude of the repulsive field. Therefore, $\nabla\rho(\boldsymbol{q})$ is equal to $\frac{\boldsymbol{q}-\boldsymbol{q}_obs}{||\boldsymbol{q}-\boldsymbol{q}_{obs}||}$.
From the total attractive and repulsive forces, the resulting one is then evaluated as:

$$\boldsymbol{F}(\boldsymbol{q}) = \boldsymbol{F}_{att}(\boldsymbol{q}) + \boldsymbol{F}_{rep}(\boldsymbol{q})$$

and it can be expressend in components, i.e. $F_x(\boldsymbol{q})$, and $F_y(\boldsymbol{q})$.
Then, the reference signals are generated as:

$$\psi_d = \arctan\frac{F_y(\boldsymbol{q})}{F_x(\boldsymbol{q})}$$

$$V_d = \frac{V_{xmax}\theta^h}{\theta^h + \Delta^h(\boldsymbol{q})}$$

The parameters of the APF has been kept the same of [4], in order to make a proper comparison between the old PID controllers and the new sliding mode one designed.

3.2.2 Navigation

This topic concerns the localization of the UGV during missions.
The initial position and orientation of the UGV is always assumed to be known. Then, during the next simulation steps, it must be determined with algorithms that exploits sensors data.
For the simulations run in MATLAB/Simulink, only odometry is used to derive the configuration of the Devastator. Recalling that the plant outputs the angular velocities of the wheel, it is possible to evaluate the longitudinal velocity and the angular velocity of the UGV with respect to the local frame. So, performing the matrix multiplication showed in chapter 2:

$$\begin{bmatrix} v_x \\ v_y \\ \psi \end{bmatrix} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{x'} \\ v_{y'} \\ \psi \end{bmatrix}$$

and integrating v_x, v_y, and ψ, the pose of the robot is known. However, exploiting only odometry relationships to obtain robot's position is not really effective for physical experiments, where an extended Kalman filter (EKF) is more suitable.
An EKF is an ideal algorithm for nonlinear models (even if it does not guarantee the optimal solution, conversely to the F applied to linear systems). The EKF used during physical testing of the controller is a ROS node provided by the navigation packages, so it has not been implemented.

3.2.3 Control

This module handles the controller (i.e., the command activity evaluation in order to follow the reference generated by the guidance block). During controller implementation, some requirements are considered, e.g., time of target reaching, robustness to model uncertainties or noise, and tracking performance.
In [4] two PID controllers were designed, however they were not so robust in simulations, even if they produced satisfying results.
In this thesis a sliding mode control approach is exploited in order to gain the desired robustness during simulations. However, the full discussion about the implemented controller will be handled in chapter 4. The outputs of the controller (i.e., v_c, and ψ_c) are then used for PWM signals evaluations:

$$PWM_L = (v_c - \psi_c)20000$$
$$PWM_R = (v_c + \psi_c)20000$$

3.3 Freedom K64F firmware

In [5], an interesting firmware structure based on PX4 Autopilot and ArduPilot has been implemented, providing fewer and simpler functionalities. In the conclusions section of [5], the possibility of adapting the designed firmware to the Devastator platform has been mentioned, and in this thesis it has been tackled.

The development of the firmware for the K64F board has been handled through Visual Studio Code and its extension PlatformIO. The next phase was the selection of an operating system, and it was the updated version of the one already used in the existing firmware, i.e., MbedOS 6. The latter operating system provides real-time functionalities, so it is a suitable one for GNC purposes.
So, once the tools were configured and the OS selected, the firmware design and implementation started, taking the code structure of [5] as a reference. Some changes has been made: GNC modules have been completely removed and hosted in the LattePanda as ROS nodes, leading to a lighter executable for the K64F board. This means that the software requirements for the K64F firmware concerned only sensing/actuation/communication functionalities.
The firmware is coded in C/C++ language and between the MbedOS APIs we can find support for:

- **Threads**: a part from the main thread, other ones can be instantiated allowing a classical *divide et impera* approach for software design (i.e., the main functionalities are divided among different threads, allowing an easier implementation). The old and the new firmware are both multi-thread, and threads synchronization is basically handled through priorities, and concurrent access to shared variables is also safe.

- **Events**: crucial to handle thread activities, since they operate on events (i.e., functions that are called back when an event is triggered). MbedOS APIs allow posting events to EventQueue objects periodically, but also cancel/stop functionalities are available.

- **File system**: the K64F board has an SD slot, which is useful to log some data or to save other important files, like the calibration ones. Luckily, MbedOS APIs allow storage device handling and mounting of the file system of the SD card (which is a FAT one).

- **Synchronization primitives**: different ones are provided by the OS, but the unique relevant for this firmware is the mutex one. It has been used to protect shared variables from concurrent read/write access.

The main components described in the following subsections are the finite state machine, the commander, the command-line interface, the communications, and the synchronized data access.
As a remark, this firmware does not include any GNC function. It simply allows sensing/actuating to the microcontroller (able to communicate with LattePanda).

3.3.1 Finite state machine

It is an abstraction useful to provide a software system state at every time instant. The FSM implemented is a deterministic one, so the state changes only with some determined inputs and conditions.
The FSM structure is almost the same of [5], but the two states handling failures are now unreachable code, since they were not implemented (so when a failure occurs, the UGV must be set again to the initial mission starting point and the firmware started again).

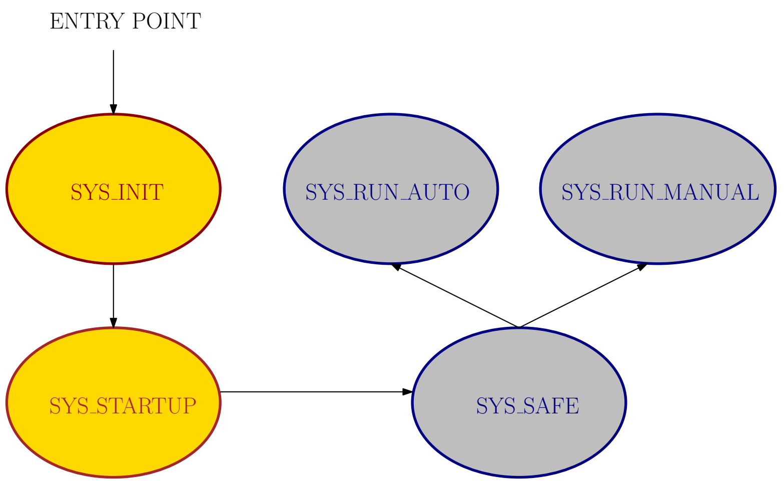


Figure 3.3: FSM of the new firmware

Figure 3.3 reports the new FSM structure:
In the FSM two states are gold coloured indicating that they handle the initialization part, while the gray coloured ones indicate states where the UGV is operative (through command-line interface, mission tackling or joystick command run).
The main activities carried in each state are:

- **SYS_INIT**: here the check for the SD card presence is performed, and, if present, the file system is mounted. After that, the presence of sensors is also checked, registering a status about them (online or not and calibrated or not). The commander module also puts the system in a disarmed condition, including a safety functionality for the PWM commands.
  Then, if the option is set, the thread which handles the log files into the SD is launched. After all those first initialization steps, the states change to the next one (i.e., SYS_STARTUP).

- **SYS_STARTUP**: here the actuation thread which handles PWM command signals is launched, making it theoretically possible to actuate motors (but it is not actually possible due to the fact that the UGV is still in a disarmed condition). Therefore the Ethernet interface is configured, launching the receiver and sender thread which handle communications with the LattePanda. In practice, sensor's data is sent (packed into Mavlink messages), and actuation command is received (also this quantity is handled through Mavlink messages, for both the joystick case and APF guidance case). At this point, the system is ready to work, so the state moves into the SYS_SAFE one, where the user can interact with the system through the command-line interface.

- **SYS_SAFE**: at this point, the command-line interface developed in [5] is available for interaction. The same commands are still available, with some small additions (e.g., the *cat* and *ls* commands, which are similar to the respective commands available in Unix terminals). Therefore, the user can choose where to start manual command through a joystick or to start a mission where the execution is automatic through GNC nodes running in the LattePanda.

- **SYS_RUN_AUTO**: in this state it is started and handled the mission specified in the APF ROS node. Practically is an autonomous driving mode in order to reach the goal point.
- **SYS_RUN_MANUAL**: here the user can perform a manual driving mode through a joystick. The joystick input is handled through a Python script over the LattePanda (which sends the command to the K64F).

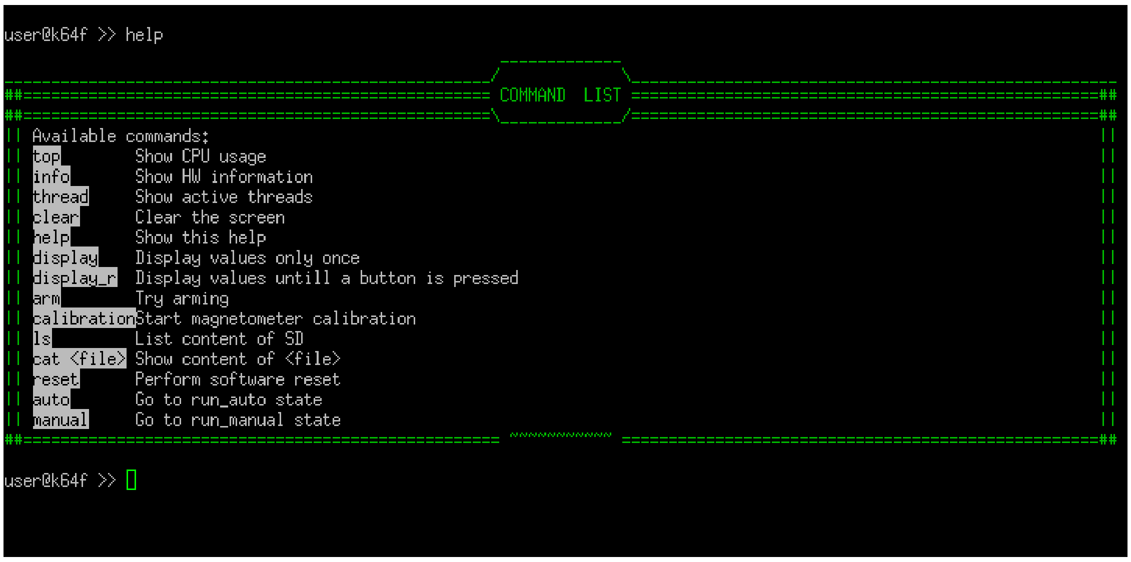
3.3.2 Commander

This module cooperates with the FSM in order to perform checks about the system state, and eventually set/get some flags concerning system state (e.g., sensors state, PWM interface activation, and consequently motor arming/disarming).
The implementation is basically the same of [5]: a C++ class that stores the system flags and provides functions to get/set such variables (i.e., getters and setters). A mutex embedded into the Read_Write_Lock class is necessary, since different threads interact at same time with the commander.
In some states, the commander performs some checks in order to ensure safety through those flags.
For the sensor components, the flag structures have been modified since different sensors are used in the UGV; therefore, communication channel states have been removed, since the Mavlink Heartbeat messages are not exploited in this firmware (an indoor UGV is not critical like a UAV, that is why heartbeat mechanism has been removed).

3.3.3 Command-Line Interface

It is still the one developed in [5], since there were already available the main functionalities necessary to interface also with the UGV. However, just for convenience in the earlier testing phases of the firmware, further commands have been added as anticipated in the previous subsection (i.e., *cat*, and *ls* commands).
One difference from the previous work, is that the CLI is always displayed in this firmware version, since it has been decided to make always available to the user to decide whether to start an automatic mission or to move the UGV through joystick. Figure 3.4 reports a screenshot of the CLI, while follows here a brief description of available commands:

- **top**: periodic display of real time CPU usage.
- **info**: shows hardware informations, OS version and available RAM.
- **thread**: lists the active threads.
- **clear**: clears the CLI window.
- **help**: shows the list of CLI commands and their respective brief description.
- **display**: prints once the sensors data.
- **display_r**:prints repeatetly the sensors data until a key is pressed.
- **arm**: an attempt to perform arming is made.



```
user@k64f >> help

                                                  ______________
--------------------------------------------------/              \--------------------------------------------------
##================================================ COMMAND  LIST ===============================================##
##================================================\______________/===============================================##
|| Available commands:                                                                                           ||
|| top        Show CPU usage                                                                                     ||
|| info       Show HW information                                                                                ||
|| thread     Show active threads                                                                                ||
|| clear      Clear the screen                                                                                   ||
|| help       Show this help                                                                                     ||
|| display    Display values only once                                                                           ||
|| display_r  Display values untill a button is pressed                                                          ||
|| arm        Try arming                                                                                         ||
|| calibrationStart magnetometer calibration                                                                     ||
|| ls         List content of SD                                                                                 ||
|| cat <file> Show content of <file>                                                                             ||
|| reset      Perform software reset                                                                             ||
|| auto       Go to run_auto state                                                                               ||
|| manual     Go to run_manual state                                                                             ||
##================================================ ^^^^^^^^^^^^ ================================================##

user@k64f >> 
```

Figure 3.4: Command Line Interface

- **calibration**: the magnetometer calibration function is started.
- **ls**: displays all filenames present in the SD.
- **cat <file>**: shows the content of the file passed as argument.
- **reset**: the firmware start-ups again.
- **auto**: enters in the automatic guide (i.e. guidance generated by APF algorithm).
- **manual**: enters in the manual mode (i.e. joystick command activity to move the UGV).

3.3.4 Communications

Sensors are connected to the K64F board through I2C communication protocol. The K64F provides an internal IMU, which provides an accelerometer and a magnetometer; an external IMU is also mounted, providing a gyroscope, and a redundant accelerometer.
IMUs and encoders data is sent to the LattePanda in order to localize the robot, generate the desired trajectory and evaluate the control action to perform the mission. The LattePanda then sends back to the K64F the control input. All those communications are handled through Mavlink v2 Messagges, which is a suitable library for GNC purposes.
In addition, those Mavlink messages are transmitted through Ethernet cable over UDP/IP. The packing of those UDP packets and their transmission is handled in the same way of the old firmware.

3.3.5 Synchronized data access

The *GlobalData* class implemented in [5] has been re-adapted: the attributes of such class have been changed according to the sensors and actuation components used by the UGV. Since the firmware is multi-thread, concurrent read/write actions on shared variables like the *global_data* one could occur, leading to the need of some synchronization patterns. In this case, the synchronization needed is the most basic one: a *mutex* variable which guarantees mutual exclusion during read/write operations. In fact, it is sufficient to protect the access on this shared variable just by calling some available functions and types provided by MbedOS.

Chapter 4

Sliding mode controller

This chapter will provide a general introduction to the sliding mode control technique in the first section. After that, the trajectory tracking problem formulation is provided and a control law is designed. In the last section, some simulation results will be shown, proving the robustness of this control strategy.

4.1 General sliding mode control concepts

"Control in the presence of uncertainty is one of the main topics of modern control theory. In the formulation of any control problem there is always a discrepancy between the actual plant dynamics and its mathematical model used for the controller design"[1]. In order to deal also with disturbances and uncertainties, robust control strategies have become relevant, and the sliding mode control technique is one of the most famous among them. Basically, in order to design a sliding mode controller, the so-called sliding surface shall be designed and then two laws are needed: the reaching law and the control law.

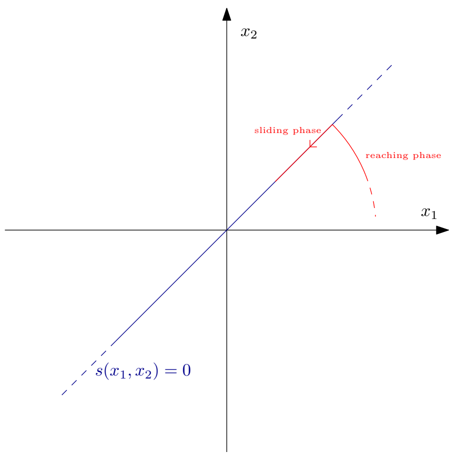


Figure 4.1: Ideal sliding mode evolution

The sliding surface is defined in the state space of the system and it is exploited to bring the system states towards the origin. The first step is thus bringing the system states into the sliding surface (i.e., the *reaching phase*) through a reaching law. Once the system states lie in the sliding surface, the *sliding phase* starts, moving system states toward the origin (i.e., the control law).
In [1] different formulations of sliding surfaces can be found. In practice, given a system described as:

$$\dot{x}(t) = \boldsymbol{f}(x) + \boldsymbol{B}(x)\boldsymbol{u}(t) + \boldsymbol{\phi}(x,t)$$

where $\boldsymbol{x}$ is the state vector, $\boldsymbol{u}$ the control input, and $\boldsymbol{\phi}$ a generic term which takes into account for disturbances. It is supposed that n states and m inputs characterize the system. In general, sliding surfaces are denoted as:

$$s(t) = \{x : \sigma(x) = 0\}$$

however, in this thesis it has been used a special sliding surface already developed in [10, 12], since with MIMO systems sliding surface design is more difficult. Details about this definition will be given in the next section.
Therefore, for both the reaching law and the control law it has been used the Gao's approach, like in [10, 12]. Among the the most relevant reaching functions it is possible to mention the *direct switching function approach* and the *Lyapunov function approach*. Instead, about the control law, a common approach is the *equivalent control* one.

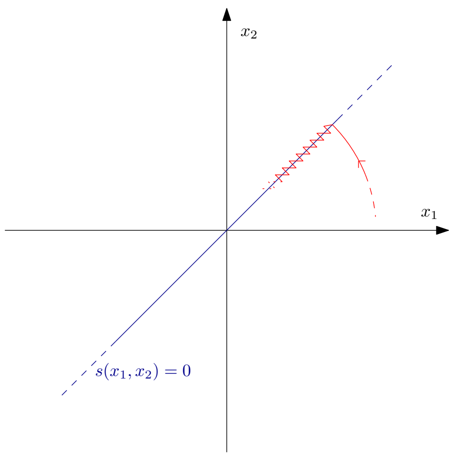


Figure 4.2: Chattering phenomenon

However, even if a good controller is designed through the sliding mode technique, a relevant and well-known problem arises during simulations: the *chattering phenomenon*. This drawback is caused by the control law based on the switching principle: in fact, in simulations and in the real world, the sliding function can not be reached perfectly (it is required an infinite frequency), and thus, due to the discontinuity of the control law, a "zigzag" behavior along the sliding surfaces arises, as shown in figure 4.2. In [1], the concept of *quasi-sliding mode* has been introduced: in this approach, the chattering is smoothed exploiting a sigmoid function instead of a discontinuous one, leading to an approximation of the sliding mode behaviour.

4.2 Trajectory tracking sliding mode controller

An interesting and effective trajectory tracking sliding mode controller is implemented based on a control law designed in [10, 11, 12].
It is first necessary to introduce the trajectory tracking problem. As suggested by its name, it consist into tracking a provided and desired reference (i.e. a trajectory), which is the output of the APF algorithm in this work.
The actual position of the robot is represented by:

$$\boldsymbol{q_r} = \begin{bmatrix} x_r \\ y_r \\ \psi_r \end{bmatrix}$$

while the desired trajectory is:

$$\boldsymbol{q_d} = \begin{bmatrix} x_d \\ y_d \\ \psi_d \end{bmatrix}$$

The objective of such trajectory tracking problem is to evaluate the error between them, which is defined as $\boldsymbol{e(t)} = \boldsymbol{x}_d(t) - \boldsymbol{x}_r(t)$, and to get:

$$\lim_{t\to\infty} ||\boldsymbol{e}(t)|| = 0$$

The APF algorithm presented in chapter 3, provides as output two reference signals, the longitudinal velocity and the orientation. However, in the trajectory tracking problem just presented we need the coordinates x_d and y_d. It is trivial to obtain them performing:

$$\begin{aligned} x_d(k) &= x_r(k) + T_s V_{ref}(k)\cos\psi_d(k) \\ y_d(k) &= y_r(k) + T_s V_{ref}(k)\sin\psi_d(k) \end{aligned}$$

where T_s is the step-size used in simulation and k the discrete time. Through this simple manipulation of outputs provided by the guidance algorithm, the problem is now well formulated and figure 4.3 gives a graphical representation of such problem.
Therefore, exploiting rotation matrices and reference frames, the error signal $\boldsymbol{e}$ can be rewritten as:

$$\boldsymbol{e} = \begin{bmatrix} x_e \\ y_e \\ \psi_e \end{bmatrix} = \begin{bmatrix} \cos\psi_r & \sin\psi_r & 0 \\ -\sin\psi_r & \cos(\psi_r & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d - x_r \\ y_d - y_r \\ \psi_d - \psi_r \end{bmatrix}$$

whose dynamics can be computed obtaining:

$$\begin{aligned} \dot{x}_e &= -v_r + v_d\cos\psi_e + y_e\omega_r \\ \dot{y}_e &= v_d\sin\psi_e - x_e\omega_r \\ \dot{\psi}_e &= \omega_d - \omega_r \end{aligned} \tag{4.1}$$

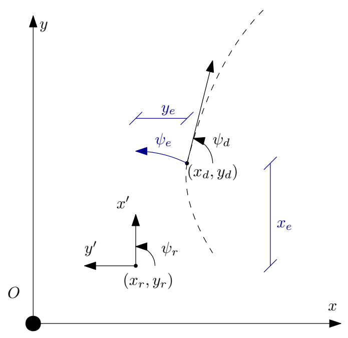


Figure 4.3: Trajectory tracking problem set-up

In order to perform trajectory tracking, an adequate control input shall be designed and in this thesis is based on SMC technique.
As anticipated previously, the first step is the definition of sliding surfaces. An interesting definition can be found in [12]:

$$\boldsymbol{s} = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} \dot{x}_e + kx_e \\ \psi_e + \arctan(v_d y_e) \end{bmatrix}$$

however, these has not been the sliding surfaces used, instead they have been:

$$\boldsymbol{s} = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} x_e \\ \psi_e + \arctan(v_d y_e) \end{bmatrix} \tag{4.2}$$

Can be noticed that the switching functions are two, since the dynamic system in equation 4.2 has two inputs.
Reachability conditions of $\boldsymbol{s}$ are satisfied [10]: for s_1 is easy to see that if it converges to zero then $x_e = 0$. About s_2 is not so obvious, but taking as Lyapunov candidate:

$$\mathcal{V} = \frac{1}{2}{y^2}_e$$

Since:

$$\dot{\mathcal{V}} = y_e \dot{y}_e = y_e(v_d \sin\psi_e - x_e\omega_r)$$

And considering that $\psi_e = -\arctan(v_d y_e)$ if $\boldsymbol{s} \to 0$, we obtain:

$$\dot{\mathcal{V}} = y_e x_e \omega_r - y_e v_d \sin(\arctan(v_d y_e))$$

Remarking therefore that for any $x \in \mathbb{R}$ and $|x| < \infty$ exists a function $x \sin(\arctan x) \geq 0$, and remembering $x_e = 0$, reachability conditions are satisfied also for s_2, since we got $\dot{\mathcal{V}} \leq 0$.
This concludes that if $\boldsymbol{s} \to 0$ we have $x_e \to 0$, which leads also to $\psi_e = -\arctan(v_d y_e)$ implying that $y_e \to 0$ and $\psi_e \to 0$ [12].
In [12, 10] the reaching law and the control law of the sliding mode are based on Gao's approach. So, the following reaching law has been applied:

$$\dot{s}_i = -q_i s_i - p_i \frac{s_i}{|s_i| + \epsilon_i}, \qquad i = 1,2 \tag{4.3}$$

with $q_i > 0$, $p_i > 0$ and $0 < \epsilon < 1$.
Reaching law 4.3 is a constant plus proportional rate one, and it already accounts for the chattering phenomenon thanks to the multiplier of p_i, which shoul be theoretically $sign(s_i)$.
The control law is derived from the reaching law 4.3, through analytical determination of $\dot{s}_i$, exploiting 4.2:

$$\begin{aligned}\dot{s}_1 &= -v_r + v_d \cos\psi_e + y_e\omega_r \\ \dot{s}_2 &= \omega_d - \omega_r + \frac{y_e\dot{v}_d + v_d(v_d \sin\psi_e - x_e\omega_r)}{1 + (y_e v_d)^2}\end{aligned}$$

Then, recalling 4.3 and solving for $\boldsymbol{u} = [v_r, \omega_r]$ we obtain the control input:

$$\boldsymbol{u} = \begin{bmatrix} v_{r=c} \\ \dot{\psi}_{r=c} \end{bmatrix} = \begin{bmatrix} q_1 s_1 + p_1 \frac{s_1}{|s_1|+\epsilon_1} + \omega_r y_e + v_d \cos\psi_e \\ \frac{q_2 s_2 + p_2 \frac{s_2}{|s_2|+\epsilon_2} + \omega_d + \frac{y_e \dot{v}_d + v_d v_d \sin\psi_e}{1+(y_e v_d)^2}}{1 + \frac{x_e v_d}{1+(y_e v_d)^2}} \end{bmatrix} \tag{4.4}$$

In order to obtain PWMs as stated in chapter 3, an integration should be performed on $\dot{\psi}_c$.
Finally, the SMC control law is thus designed for the target UGV, and in the next section suitable values of controller parameters p_i, q_i and ϵ_i will be provided, showing their effectiveness through some simulation scenarios run on MATLAB/Simulink.

4.3 SMC design and simulations

In MATLAB/Simulink two missions have been simulated, which are the same ones on which testing of already existing PID controllers has been performed in [4].
By trial and error, the following parameters in equation 4.4 has been set and used in all the reported simulations:

Parameter	Value
q_1	0.90
p_1	0.77
ϵ_1	0.30
q_2	0.01
p_2	0.50
ϵ_2	0.90

Table 4.1: Controller's parameter values

Therefore, in the control input $\boldsymbol{u}$ a saturation of 0.45 has been set on first component, while 0.10 has been set for the second one.

4.3.1 Obstacle free path

In this first mission, the robot has a starting configuration $\boldsymbol{q_0} = [0 \quad 0 \quad 0]$ and should reach the final one $\boldsymbol{q_g} = [1 \quad 0 \quad -]$. The last component of the goal vector is not relevant, since it is only required to reach the specified coordinates x and y with a tolerance of 0.1 metres.

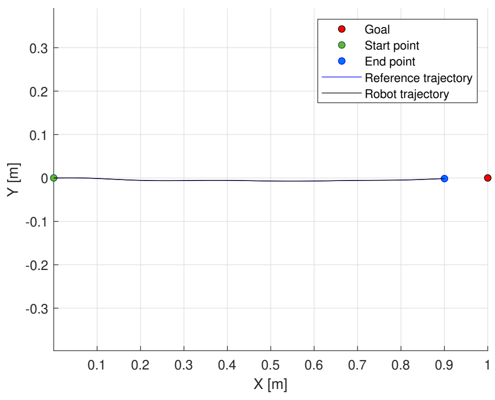
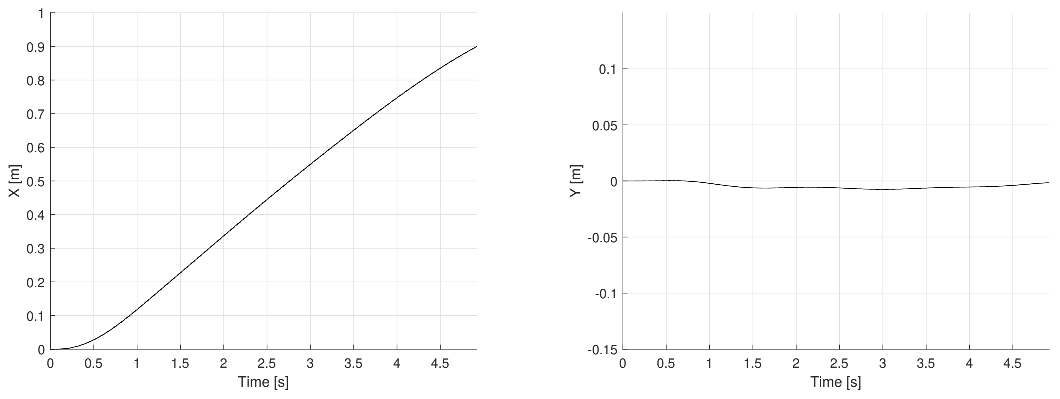


Figure 4.4: Obstacle free Trajectory

Figure 4.4 shows the resulting route performed in this first simulation. It can be noticed that the behaviour is really good, since the robot performs an almost straight line as desired. In more details, figure 4.5 explains what is meant with "good behaviour": the x coordinate is increasing almost linearly while the y one is almost zero, with a maximum error less than 0.05 metres. Are also so interesting to analyze the results of the longitudinal velocity and orientation signals, reported in figure 4.6.

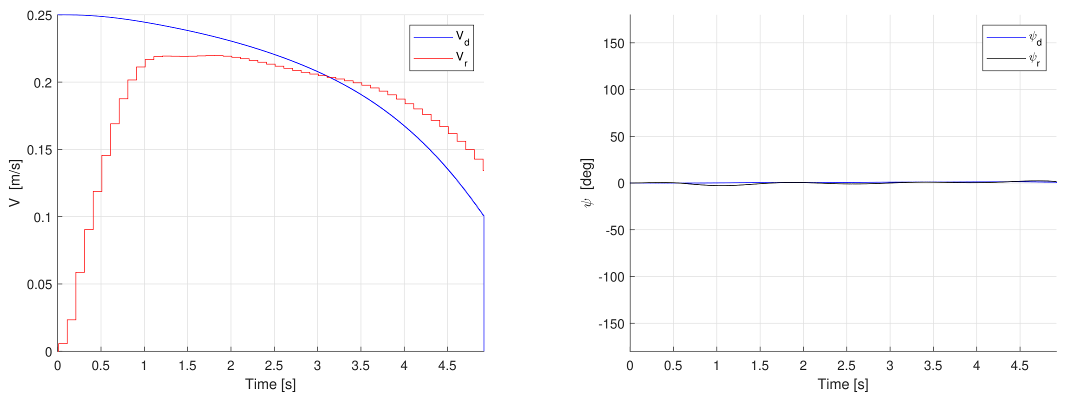


(a) X position

(b) Y position

Figure 4.5: X and Y positions

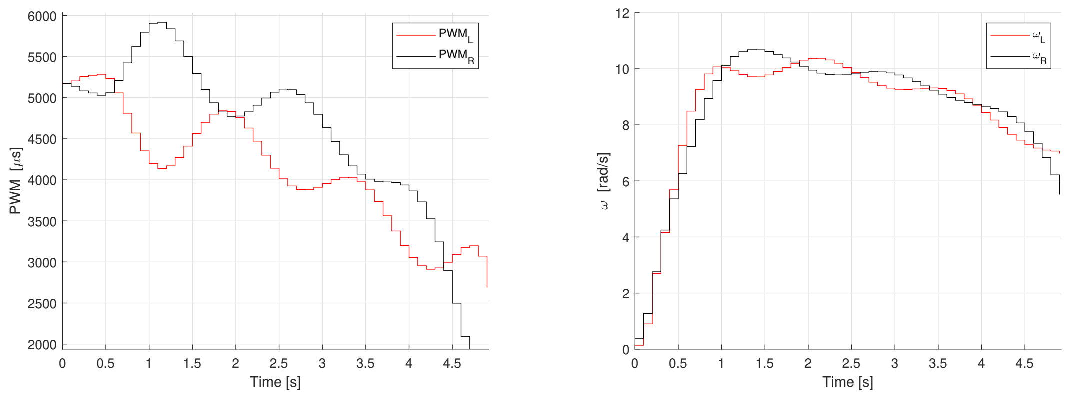
The reference orientation signal is tracked faithfully, however, the longitudinal speed is not tracked in a really good manner, even if the mission is performed correctly. Qualitatively, the reference velocity signal is tracked but not with the same accuracy of the orientation. The input/output data of the NARX block recorded in this simulation are instead reported in figure 4.7. It is interesting to notice here the asymmetry of those signals: the SMC is able



(a) Longitudinal speed (b) Orientation

Figure 4.6: APF output tracking

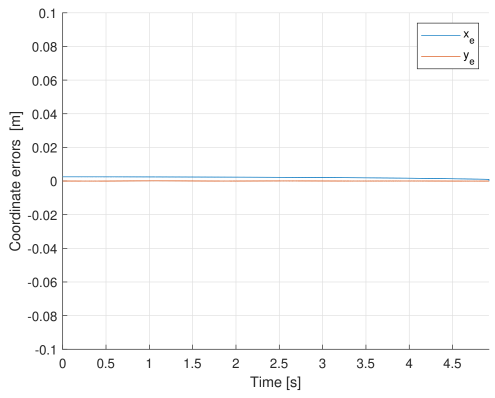
to compensate strong nonlinearities and also the asymmetric behaviour observed during data collection phase in the system identification process.



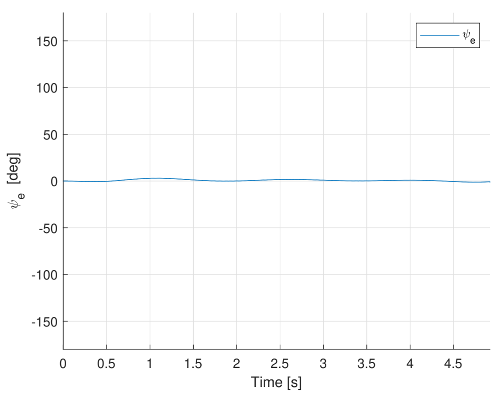
(a) PWM signals (b) Wheels angular velocities

Figure 4.7: Input/output data

In addition, about the controller, is interesting to observe the evolution of the state variables used to design the controller (i.e. x_e, y_e and ψ_e). Figure 4.8 reports those variables: the coordinates are tracked in a really accurate way and so the error of x and y are almost zero, but also the heading has an overall good accuracy (some oscillations occur, but they are negligible).
Therefore, the last interesting observation can be made on the plot of the sliding surfaces, reported in figure 4.9: the first one, since is defined as the error on x coordinate, is almost zero. On the other side, the second sliding surface shows more inaccuracy, and this is due to the oscillations on ψ_r signal.



(a) X and Y coordinate errors



(b) Orientation error

Figure 4.8: Controller error variables

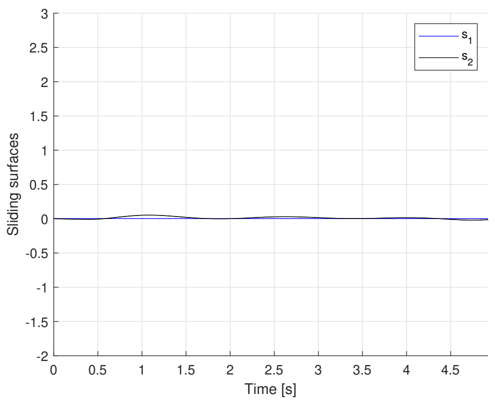


Figure 4.9: Sliding surfaces

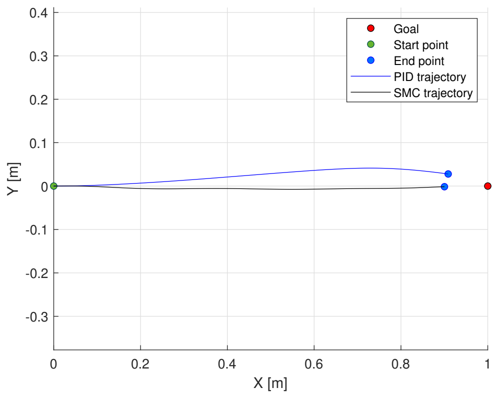


Figure 4.10: Controllers comparison

Concluding, we can say that this mission was successful, except the longitudinal velocity tracking. Therefore, the chattering phenomenon is not so visible in those simulations (but present, as proved by ψ_r).
It is interesting to notice the robustness reached with this control law: in figure 4.10 the trajectory performed with the old plant model and old PID controllers is reported against the new controller and plant: it is undoubtedly evident that the robust feature has been achieved. The simulation time to perform such mission is almost the same for both them, but the sliding mode controller has a more regular pathway toward destination.

4.3.2 Environment with obstacles

In this mission, the robot has once more the starting configuration $\boldsymbol{q_0} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$ and should reach the target one $\boldsymbol{q_g} = \begin{bmatrix} 6 & 3 & - \end{bmatrix}$. The last component of the goal vector is again not relevant, since mission requirements are still the same. However, in this scenario 4 obstacles have been introduced, which have coordinates $x = \begin{bmatrix} 2 & 3 & 4 & 5 \end{bmatrix}$ and $y = \begin{bmatrix} 3.5 & 1 & 2.5 & 3 \end{bmatrix}$. In this case, we can observe in figure 4.11 that the UGV successfully

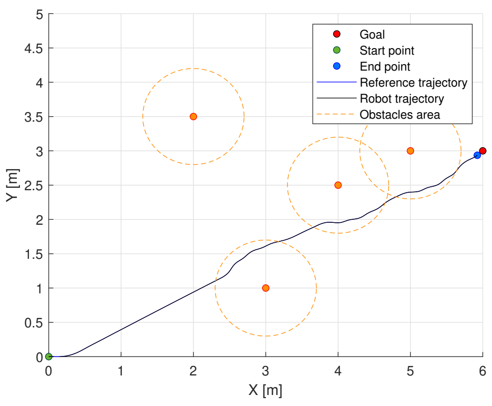
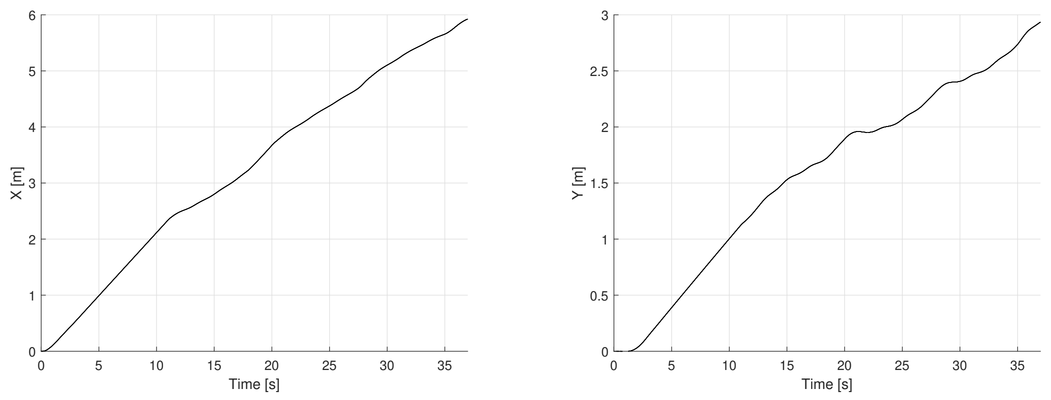


Figure 4.11: Trajectory with obstacles

goes into goal's direction and overcome obstacles with good results. More in details, the coordinates behaviour are shown in 4.12: they are both almost linearly increasing, which indicate a path that should be approximately straight. Again, referencing to figure 4.14, the longitudinal velocity tracking performance is not as good as the orientation tracking one, which has an overall acceptable behaviour even if not accurate as the first mission.
About input/output signals, in this case we can still notice the presence of asymmetry in figure 4.13, which means again that real uncertainties and nonlinearities are taken into account succesfully, since the tracking performances on x and y coordinates are exceptional.

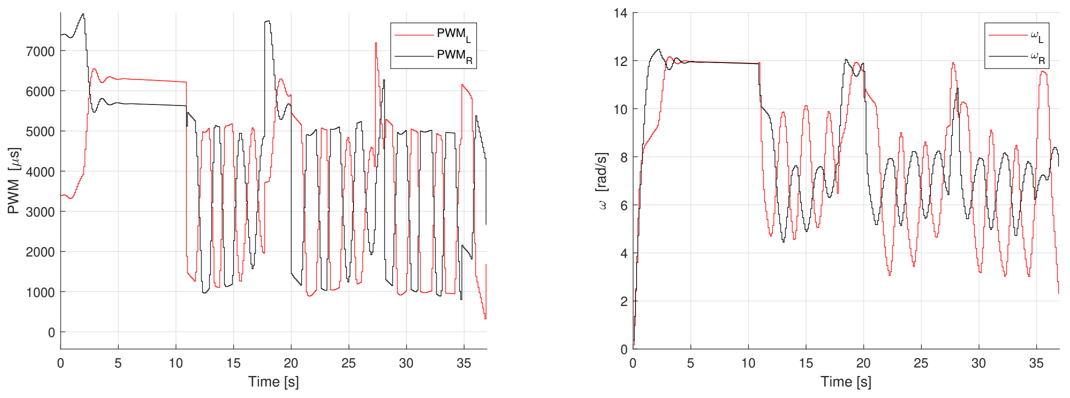
Is now interesting to observe the behaviour of the state variables x_e, y_e and ψ_e. Figure 4.15 shows that the errors are almost zero, so the coordinates are tracked again in an excellent way. The heading has also an overall good accuracy, except the instances where an obstacle avoidance manoeuvre occurs.

At last, the plot of the sliding surfaces in figure 4.16: for the first one still holds the previous considerations (i.e., since is defined as the error on x coordinate, s_1 is almost



(a) X position (b) Y position

Figure 4.12: X and Y positions

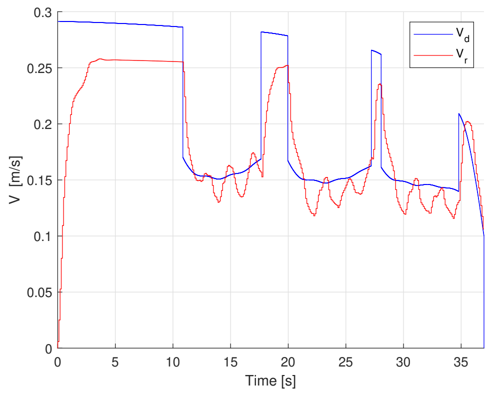


(a) PWM signals (b) Wheels angular velocities

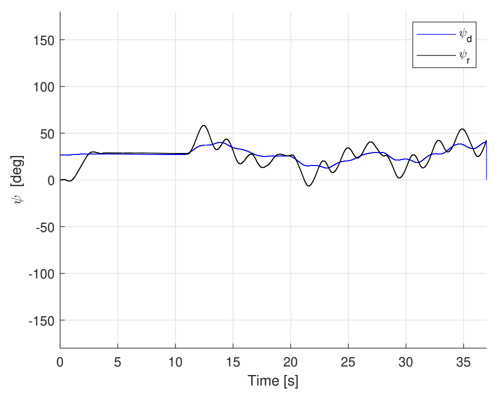
Figure 4.13: Plant input/output data

zero). On the other side, the second sliding surface shows more inaccuracy, according to ψ_e signal.

Ending, this mission was success too, a part from the longitudinal velocity and the orientation signal during obstacle avoidance manoeuvres. It is also interesting to compare again the performances with the old system: in figure 4.17 such comparison is reported. It is clear that the SMC tends to point toward the goal point in a more robust way with respect to the PID controllers, and this feature could lead to lesser time taken to tackle missions (in particular obstacle configurations, with the shown simulations the time to complete missions was really similar).

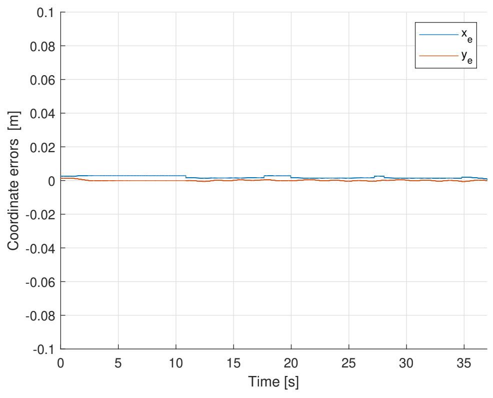


(a) Longitudinal speed

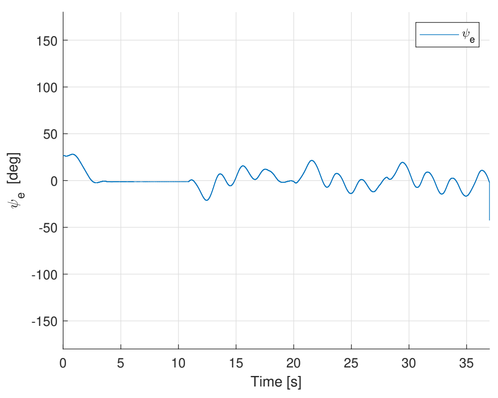


(b) Orientation

Figure 4.14: APF output tracking



(a) X and Y coordinate errors



(b) Orientation error

Figure 4.15: Controller error variables

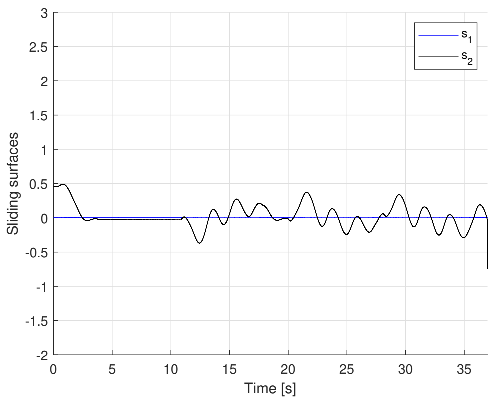


Figure 4.16: Sliding surfaces

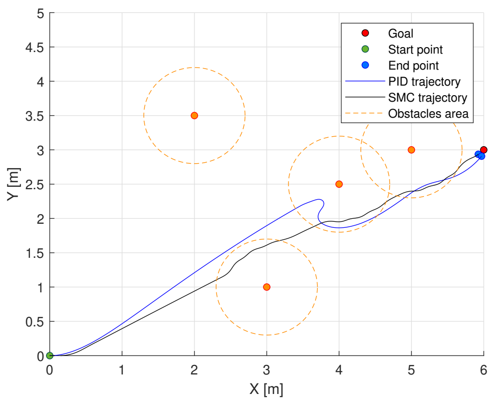


Figure 4.17: Controllers comparison

Chapter 5

Experimental simulations

In order to make laboratory simulations with the designed controller, ROS framework has been used. Some basic concepts about ROS are illustrated here in next section, and in appendix A the controller code (i.e., a ROS node written in Python).

5.1 ROS basics

"ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management."[16]
First basic functionalities reported on ROS documentation are about the filesystem level: some useful functions are provided in order to organize in a professional and convenient way the source code. In fact, ROS projects are organized in packages and filesystem functions permit to find packages, navigate and of course create them. A further advantage, is that those functions are quite similar to the ones in Unix environment, so it is really easy to learn basic functionalities for someone which is familiar with Unix environments.
For example, among the most relevant filesystem functions we can find:

- *rospack find <package_name>*: used to find the location of the package passed as argument.
- *roscd <package>*: used to enter into the directory of the package passed as argument.
- *rosls <package>*: list the files of the package provided as argument.

A workspace can be created through *catkin_create_pkg* command, and then it can be build with suitable commands; every package has also some dependencies, and the *rospack depends1 <pkg_name>* command is used to show them.
After the setup and the first build of a package, the development of ROS nodes can start. Basically, the ROS node concept is quite similar to the one of executable programs: indeed, it contains C++/Python code which is written in order to perform some operations, and suitable tools are provided in order to enable communication between nodes (i.e. topics). Development through nodes allow an easy design of the overall system, since tasks can be

divided into simpler ones, and in case of failures troubleshooting activity becomes more immediate and intuitive. Before running a developed ROS node, the *ROS master* node shall be launched (using *roscore* command). After that, the desired nodes can be launched in order to perform desired computations.
Each node can publish the computed data into special communication channels provided by ROS, i.e., the topics. On the other side, other nodes could subscribe to them in order to receive some data. The data written into topics are called *messages*, and they can be personalized in in the workspace (some basic ones are already provided).
Given this structure, an intuitive way to have a representation of the system architecture is using graphs. In effect, the *rosrun rqt_graph* command is able to plot a graph containing running nodes and the communications occurring between them, specifying also the topics names.
Therefore, during simulations, can be useful the *rosbag record <topic_names>* command to save the data sent over topics. Then, saved data can be processed for plots/reports, but also a virtual representation of the simulation can be shown through Rviz.
The ones showed above, are the basic concepts used to perform real world simulations through ROS. However, this was just a very brief and basic introduction on ROS.
Finally, in appendix A, the source code of the implemented SMC node is reported.

5.2 Real mission results

As in [4], a simple mission without obstacles with the real UGV has been performed. However, this time the mission differs and consists in reaching the coordinates $x = 1$, $y = -1$ starting with the usual initial configuration $\boldsymbol{q_0} = [0 \quad 0 \quad 0]$.

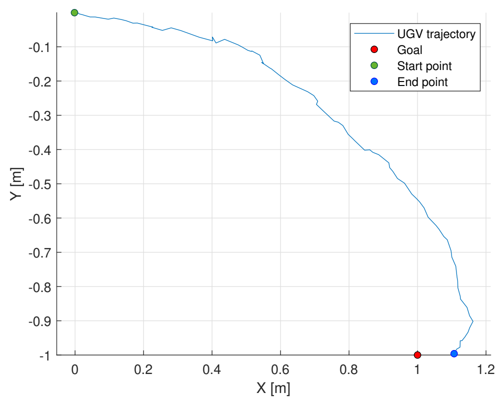
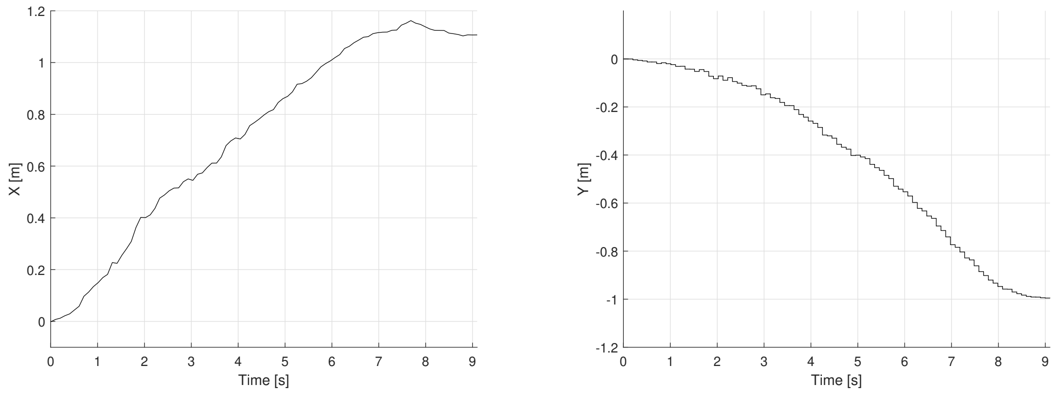


Figure 5.1: Real UGV Trajectory

In this case the mission is again completed, however the results are not so good with respect to the ones expected in simulation environments. In fact, comparing the simulation trajectory with real trajectory in figure 5.4 we can immediately notice that the real one has a worse behaviour.



(a) X position (b) Y position

Figure 5.2: X and Y positions

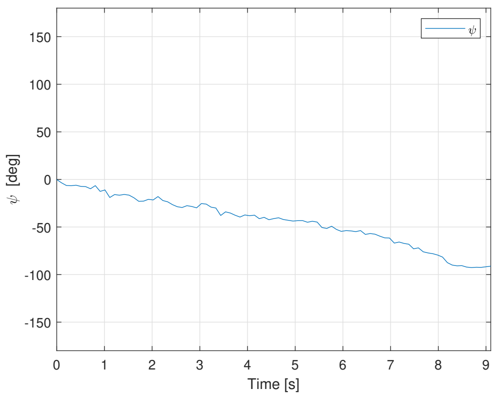


Figure 5.3: Real UGV orientation

Probably, a new parameters set for the controller should be determined by trial and error tuning (they were the same of the simulations). Another motivation could be the calibration of sensors, probably it should be repeated in a more precise way to increase the performances of the EKF; however this is not the most relevant cause of those performances, in fact the orientation signal has a coherent shape with the plotted trajectory. Therefore, also the APF could need some tuning for the real world experiments (it had the same parameters of simulations). On the other hand, in simulation environment the UGV showed as usual a behaviour where it goes straight to the goal, which is another experiment confirming good performances. Therefore, the simulation time was about 7 s, while in the real experiment it was slightly higher, 9.1 s.

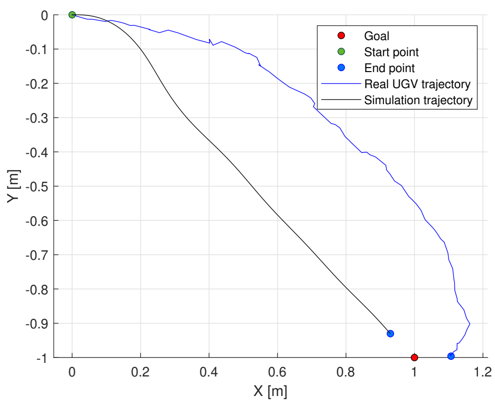


Figure 5.4: Simulated and Real trajectories comparison

Conclusions

This thesis began as a further work of [4], with the main objective of a robust controller design. However, a re-fine of the previous model has been necessary and a new firmware structure implemented.
About the NARX model, even if the results were generally acceptable, it is clear that a modeling approach based only on kinematics is a quite limiting one. This approach, excluding the longitudinal velocity tracking, provided really interesting simulations results, but the real experiment conducted proved that a dynamical modeling could be necessary to reduce the differences between simulations environment and the real one. It may also be necessary a tuning of the APF algorithm parameters, and also trying to change the controller's ones used on the physical UGV: using the same of simulations is rarely the right choice, it is common to change those parameter but an attempt to keep the same on physical simulations has been made after the good results on MATLAB/Simulink simulations.
Regarding the controller, it is proven by simulation results that the desired feature of robustness has been reached in an excellent manner, and the tracking of the errors used to generate the control input is really good. However, it is quite problematic the tracking of the longitudinal velocity provided by APF algorithm: this controller is worse than the previous one on this, but on the other hand it is really better on heading tracking. Therefore, as proven in all simulations shown, the robot tends to go through the goal with a straight motion in obstacle free environment, which is the main result of this thesis.
Sadly, the features of simulations provided by this sliding mode control approach were not reached in the real experimentation. It would be necessary to tune togheter controller's and APF's parameters; as a further work, could be also interesting the adoption of this proposed controller with a plant modelled through dynamical equations and not only an identified one.
Instead, the implementation of the new firmware structure has been really successful. As anticipated in previous chapters, it has the main advantage of allowing modularity of every part, it is much more easy to interpret thanks to states introduction and it is light for the K64F. However, as a drawback, there is for sure the absence of error handling, which corresponding states have not been used. Error handling is really tricky, but since the main goal was the robust controller design and the target UGV has not safety requirements, the development of error recovery functionalities has been left incomplete and it could be another further work.
Concluding, methodologies shown in this thesis can be considered an interesting approach for prototyping mobile robots and algorithms: the identified plant model can handle strong nonlinearities as proven in this thesis, the new firmware and ROS nodes structures makes

the software architecture really modular which leads to the possibility of testing different algorithms for the guidance, navigation and control subsystems.

Appendix A

Appendix A

Is reported here the ROS node which implements the sliding mode controller. It has been translated manually into this code and tested in Simulink through Python function call inside a MATLAB function block. The output of this code and the block diagram of the controller subsystem were the same for different simulations, proving that any bug should not be present in this node.
Code generation could be a good alternative but it has been excluded for this thesis project.

```python
from numpy import linalg as LA
import numpy as np
import math
import queue
import rospy
import message_filters
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Pose
from sensor_msgs.msg import Imu
from visual_odometry.msg import PWM_cmd,APF_cmd
from scipy.spatial.transform import Rotation as R



class SMC():


    def __init__(self):
        rospy.init_node('SMC',anonymous=True)
        self.rate = rospy.Rate(10) # 10hz
        sub_APFout = message_filters.Subscriber("/APF_output", APF_cmd, queue_size = 10)
        sub_goal = message_filters.Subscriber("/goal", Pose, queue_size = 10)
        sub_odometry = message_filters.Subscriber("/odometry/filtered", Odometry, queue_size = 10)
        self.PIDpub = rospy.Publisher('SMC_cmd', PWM_cmd, queue_size=10)
        self.msg = PWM_cmd()
        self.psid_prec = 0
```

```
       self.vd_prec = 0
       ts = message_filters.ApproximateTimeSynchronizer([sub_APFout,
        ↪  sub_odometry, sub_goal], queue_size=10, slop=0.5,
        ↪  allow_headerless=True)
       ts.registerCallback(self.ctr_step)
       rospy.spin()

    def get_rotation(self,Odom):
        orientation_q = Odom.pose.pose.orientation
        orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z,
         ↪  orientation_q.w]
        r = R.from_quat(orientation_list)
        EuAn = r.as_euler('zyx', degrees=False)
        return EuAn


    def ctr_step(self,sub_APFout, sub_odometry, sub_goal):


        xd_dot = sub_odometry.twist.twist.linear.x
        v_y = sub_odometry.twist.twist.linear.y
        xr = sub_odometry.pose.pose.position.x
        yr = sub_odometry.pose.pose.position.y
        wr = sub_odometry.twist.twist.angular.z
        vd = sub_APFout.Vref

        psid = sub_APFout.Psiref
        wd = (psid-self.psid_prec)/0.1
        vd_dot = (vd-self.vd_prec)/0.1
        xd = xr + vd*0.1*math.cos(psid)
        yd = yr + vd*0.1*math.sin(psid)
        [psir, _, _] = self.get_rotation(sub_odometry)
        vr = (xd_dot+v_y)/(math.cos(psir)+math.sin(psir))


        #errors definition
        xe=math.cos(psir)*(xd-xr)+math.sin(psir)*(yd-yr)
        ye=-math.sin(psir)*(xd-xr)+math.cos(psir)*(yd-yr)
        psie=psid-psir
        if math.fabs(psie) > math.pi:
            psie=psie-2*math.pi*np.sign(psie)

        # smc - surfaces
        s1=xe
        s2=psie+math.atan(vd*ye)

        # smc - command evaluation
        eps1 = 0.3
        eps2 = 0.9
        u1=0.9*s1+0.7*s1/(np.abs(s1)+eps1)+wr*ye+vd*math.cos(psie)
        u2=(0.01*s2+0.5*s1/(np.abs(s2)+eps2)+wd+(ye*vd_dot+vd*vd*math.sin(psie))/(1+(ye*vd)*(ye*vd)))/(1+xe*vd/(1+(ye*vd)*(ye*vd)))

        if u1>0.45:
            u1 = 0.45

        if u1<0:
            u1=0

        if u2 > 0.1:
            u2 = 0.1

        if u2 < -0.1:
            u2 = -0.1

        PWM_R = 20000*(u1+u2)
        PWM_L = 20000*(u1-u2)

        if PWM_R>20000:
            PWM_R = 20000

        if PWM_R<-20000:
            PWM_R = -20000

        if PWM_L>20000:
            PWM_L = 20000

        if PWM_L<-20000:
            PWM_L = -20000

        self.psid_prec = psid
        self.vd_prec= vd

        self.msg.PWM_left = PWM_L
        self.msg.PWM_right = PWM_R
        self.PIDpub.publish(self.msg)


if __name__ == '__main__':
    try:
        SMC()
    except rospy.ROSInterruptException:
        pass
```

Bibliography

[1] Shtessel Y., Edwards C., Fridman L., Levant A., *Sliding Mode Control and Observation*, 2013.

[2] Siciliano B., Sciavicco L., Villani L., *Robotica. Modellistica, pianificazione e controllo*, 2008.

[3] MathWorks,`https://it.mathworks.com/help/ident/gs/about-system-identification.html`.

[4] Incoronata Trombetta E., *Identification Methodsand Simulation Modeling of a small UGV for Indoor Applications*, Politecnico di Torino, 2021.

[5] Sala Y., *Firmware Design and Implementation for a Quadrotor UAV*, Politecnico di Torino, 2021.

[6] Regruto Tomalino D., *Modeling and control of cyberphysical systems*, Course Notes, Politecnico di Torino, 2021.

[7] Lennart L., *System Identification*, Wiley Encyclopedia of Electrical and Electronics Enginnering, 2017.

[8] B. Raafiu, P. A. Darwito, *Identification of Four Wheel Mobile Robot based on Parametric Modelling*, International Seminar on Intelligent Technology and Its Apllications, 2018.

[9] DFRobot, `https://www.dfrobot.com/product-1477.html`.

[10] Lu Yang and Shenghui Pan,*A Sliding mode control method for trajectory tracking control of wheeled mobile robot*, J. Phys.: Conf. Ser. 1074 012059, 2018.

[11] Capello E., Locardi D., Arbinolo F., Sartori D., Ermacora G., Guglieri G., Yu W.,*Modelling and control of a skid-steering mobile robot for indoor trajectory tracking applications*, submitted.

[12] Locardi D., *Modelling and Control of a Skid-Steering Mobile Robot for Indoor Trajectory Tracking Applications*, Politecnico di Torino, 2020.

[13] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert, *A review of mobile robots: Concepts, methods, theoretical framework, and applications*, International Journal of Advanced Robotic Systems 16.2 (2019).

[14] Henrik Andreasson, Giorgio Grisetti, Todor Stoyanov, Alberto Pretto, *Sensors for Mobile Robots*, 2022

[15] Mars 2020 `https://mars.nasa.gov/mars2020/`

[16] ROS, `http://wiki.ros.org/ROS/Introduction`