POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Integration of Matlab-based controllers with ROS for autonomous agricultural vehicles

Supervisors

Ing. Fabrizio DABBENE

Dott.ssa Martina MAMMARELLA

Dott. Antonio PETITTI

Candidate

Gerardo DETTA

DECEMBER 2022

Summary

In agriculture automation, Robot Operating System (ROS) is one of the platforms mainly used for robot software development, thanks to its capability of integration with other tools. In particular, ROS offers the versatility of implementing control algorithms in different programming languages. The traditional approach is to use directly Python or C++. In this work, the integration between ROS and Matlab has been investigated, to explore the possibility to combine the potentialities of both environments: versatility and re-usability of ROS, and richness of tools, especially in control systems, for Matlab. In particular, this thesis focuses on the comparison of control performance for an autonomous ground vehicle achieved implementing the same control laws, i.e. a PID and a LQR, using different programming languages. In details, C++, Python, Matlab, and C++ obtained from Matlab coder have been investigated, comparing their performance in terms con trajectory tracking and execution time. As simulation environment, the Gazebo simulator, integrated with ROS, has been exploited. The results suggest that Matlab tools and support functions add a great contribution to ROS, simplifying the controller development and also helping to get a better optimized code. Moreover, using C++-compiled Matlab code also allows to significantly reduce the computational burden.

Acknowledgements

I would first of all thank my thesis supervisors Ing. Fabrizio Dabbene, Dr.ssa Martina Mammarella and Dott. Antonio Petitti for the great opportunity to work on these topics, the continuous support and understanding in the most difficult moments.

I'm extremely grateful to my family, my parents, my brother Antonio, my grandparents, my aunts and uncles, for supporting me during these years and never making me miss anything. I remember five years ago, getting on the train to Turin for the first time, with suitcases full of food, jars, fears and hopes. Now a first milestone has been reached, but it is only the start. However, I have more certainties than in the past, because I know that I will always bring with me your teachings.

I would like to thank my long friends, particularly Michele, Antonio and Italo, who have been physically far, but have always supported me.

This endeavor would not have been possible without the family of (S)Quarto Piano, that welcomed me when I moved to Turin, and has become a second home for me. Living in this dynamic environment gives me serenity, being able to focus on the studies, but also encouraged me to grow, learning from many interesting people.

I do not want to forget also all the other people met in Turin during these 5 years, they too have become part of my life, and helped me to become what I am now.

"Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning"
Albert Einstein

Table of Contents

List of Tables

List of Figures

Acronyms

ROS Robotic Operating System

PID Proportional Integral Derivative

LQR Linear Quadratic Regulator

UGV Unmanned Ground Vehicle

DDMR Differential Drive Mobile Robot

RDE Robotic Development Environment

CoG Center of Gravity

MIMO Multiple-Input Multiple-Output

ODE Open Dynamic Engine

URDF Unified Robotic Description Format

SDF Simulation Description Format

LTS long-term Support

Chapter 1

Introduction

1.1 Mobile robots in agriculture

In recent years there has been an increasing trend in the usage of autonomous robots in agriculture. Several reasons are behind the evolution of agricultural work, i.e. the growth of global population, that caused an increment of production levels of primary goods, the reduction of human impact on environment, with the necessity of maximising the harvest using smaller portion of land, the reduction of pesticides usage or the lack of workers in this sector, as described in [1].

Autonomous robots can represent a solution to these problems thanks to their potential in terms of repeatability, accuracy and durability, as reported in [2], being able of increasing the production efficiency, improving its quality and reducing the production time.

The most common autonomous robots for agriculture purposes are the Unmanned Ground Vehicles (UGV) that are used to move within fields and performing farming tasks along the route, i.e. harvesting, weeding, disease detection and seeding, as explained in [3]. From these examples it can be deduce that path tracking is one of the main functionalities to be provided by UGVs in agriculture.

However, robot's potential in agriculture is not yet fully exploited, since there are serious challenges to overcome, i.e. agriculture environment not well structured, low grip conditions, problems in sensor data acquisition and localization, that cause low accuracy in estimating the orientation and speed of the vehicle. This limits their usage in high-precision tasks and also in human-machine interaction, due to safety reasons. Trying to solve these issues with conventional methods is not suitable in most of the cases, since it requires high technological knowledge and high costs.

Some Robotic Development Environments (RDE) have been proposed to potentially solve these problems, as stated in [4] and [5]. The Robot Operating System (ROS) is one of the most popular one, thanks to its many great features, such as distributed computing, software sharing and reusability, hardware support and integration with other platforms. ROS gives the opportunity to use a vastness of libraries of open-source software regarding the main fields of robotic research, like navigation, control and sensor data processing, so it provides a source of knowledge based on past experience, and solutions to common problems, that can be shared, without "reinventing the wheel each time". This allows to provide a reliable and cost-effective solution to the agriculture automation challenges.

It is possible to find in literature some examples of ROS integration in agricultural robotics, like the results of CROPS project [6] reported in [5], which aims at developing robots for agriculture and forestry use. In this project, most of software development has been carried out in ROS, using C++ as programming language, and it has been used for many subsystems, like sensing, perception, and mission control. One problem emerging by the usage of ROS is that it is not a real-time framework, while for robot motion, is very common to use real time systems. There are many approaches to integrate real time processes with ROS, and the solution chosen for the CROPS project is a real-time control unit, running the commercial operating system xPC Target [7], integrated with Matlab/Simulink, that handles control and low-level communication with robot's actuators. It communicates with ROS using the Matlab integration, thanks to the ROS Toolbox [8], that is fully supported and with good documentation.

Another good reason for integrating ROS with Matlab is the richness of tools in data processing and control system design that the latter provides, and that allows to simplify the robotic system design. In [9] also another advantage of the ROS Toolbox has been analysed, that is the implementation of a ROS node directly inside the Matlab environment, even using a Windows operating system, increasing the possible applications.

1.2 Thesis goals and objectives

This thesis has the goal to compare the control performances of some control laws for trajectory tracking of an UGV, using different programming approaches. In particular, the classical approach used with ROS, that involves the usage of C++ or Python for programming nodes was compared with a Matlab-integrated approach, in which the control algorithm is designed in the Matlab environment, tested, using the integration with ROS, and then exported in a C++ stand-alone node. Then, the results are compared, focusing mainly on trajectory tracking performances and execution time.

As UGV, a Differential Drive Mobile Robot (DDMR) has been used, i.e. the TurtleBot3 package, taking advantage of one of the libraries offered by ROS. This is one of the most used platforms for ROS among developers and students [10], offering extensive documentation.

About controllers, two different techniques have been designed and tested. i.e. a PID and LQR, since they are two of the most used approaches in trajectory tracking for DDMR, as shown in [11] or [12].

To test the different approaches, Gazebo has been used as simulation environment, being the major graphical interface of ROS.

1.3 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 focuses on the design of the DDMR model and the theoretical background about PID and LQR controllers. Chapter 3 provides an introduction to ROS and its integration with Matlab. It is conceived in the form of a manual, containing the researches and experience gained with this work. In Chapter 4, the simulation setup is outlined, with the definition of the reference trajectories used for the path tracking, the controllers' implementation and tuning, the Gazebo setup, and result's analysis. In Chapter 5, conclusions are drawn with a focus on future works.

Chapter 2

Theoretical background

This chapter presents the theory used for the trajectory tracking controllers design.

The first section introduces the mathematical model of the differential-drive robot: we start from the wheel kinematics to obtain the unicycle model.

Finally, the last two sections are dedicated to the PID and LQR controllers, describing the mathematical theory and the discrete time implementation.

2.1 Mathematical model

The system considered for this work is a Differential-Drive Mobile Robot (DDMR), that is a mobile robot that moves thanks to two controllable wheels. If the relative rate of rotation of the wheels is the same, it moves in straight direction, while varying the relative rotational velocity it is able to change direction. For stability purposes, an additional castor wheel is also added, which is an undriven wheel that turns in the movement direction.

2.1.1 Differential-Drive Model

To describe the motion of these types of robots, it is possible to use the differential-drive model described in [13] and represented in Fig. 2.1.

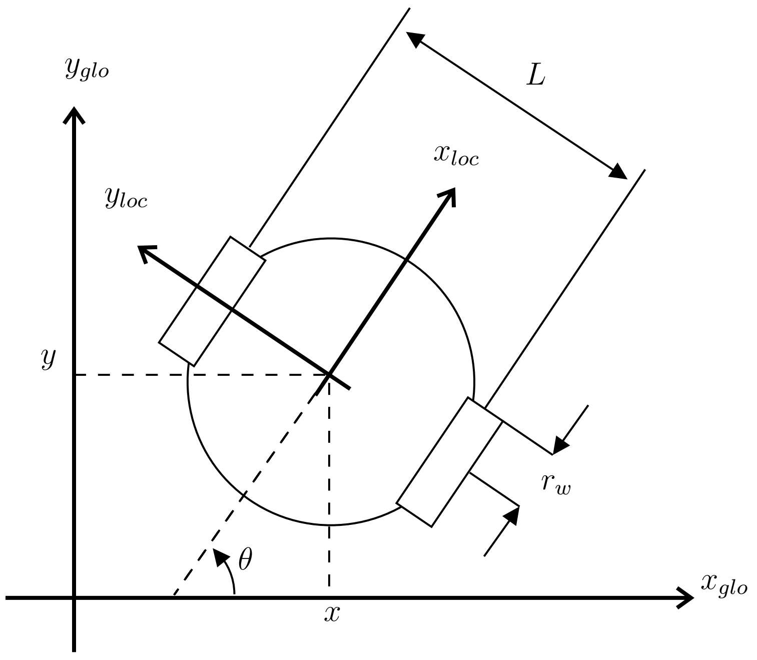


Figure 2.1: Symbols for differential-drive model

To represent the robot model, two sets of Cartesian reference frames are considered: the local, or body, reference frame $f(x_{loc}, y_{loc}, z_{loc})$ and the global, or fixed, reference frame $f(x_{glo}, y_{glo}, z_{glo})$. The local frame is a non-inertial reference system, originated from the robot center of gravity (CoG), with the x axis aligned with the robot forward direction, while the y axis points on the left. Together with the z axis exiting from the x-y plane, they form a right-handed reference frame. The global reference frame is an inertial frame, that has its origin in a fixed point of the space. It is possible to represent a vector defined in the local frame into the global frame through a rotation about z_{glo} axis of angle θ and the rotation matrix that relates the two reference frames is defined as

$$R_{\text{loc,glob}} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

The differential-drive model is based on the kinematic of the wheels, assuming that there is no slippage and the surface is perfectly flat. Hence, we have that the linear velocity of one wheel v_w is defined as

$$v_w = \omega_w r_w \tag{2.2}$$

where ω_w is the wheel angular velocity and r_w is the wheel radius.

Since the two wheels can rotate with different velocities, i.e. ω_L (left wheel) and ω_R (right wheel), then the velocity v of the robot CoG can be computed as

$$v = \frac{(\omega_L + \omega_R)}{2} r_w \tag{2.3}$$

Moreover, having that the surface is perfectly flat, it is possible to consider the robot motion in a 2D Cartesian plane, i.e. robot position is described by x and y coordinates in global frame, such that the two components of the linear velocity of the robot along the x and y axes are defined with respect to the robot orientation θ as

$$\dot{x} = \frac{(\omega_L + \omega_R)}{2} r_w \, cos\theta \tag{2.4}$$

$$\dot{y} = \frac{(\omega_L + \omega_R)}{2} r_w \, sin\theta \tag{2.5}$$

Moreover, we have that the angular velocity of the robot ω, expressed in the fixed frame, is related to the difference of angular velocities of the wheels, i.e.

$$\omega = \dot{\theta} = \frac{(\omega_R - \omega_L)}{L} r_w \tag{2.6}$$

where L is the distance between the wheels.

From reference frame definition, angles are measured counter-clockwise from x axis. In this way, if ω_R is higher than ω_L, the angular velocity is positive, since the orientation angle is increasing.

2.1.2 Unicycle Model

Starting from the differential-drive model and substituting expression (2.3) in equations (2.4) and (2.5) we have

$$\dot{x} = v\ cos\theta \tag{2.7}$$

$$\dot{y} = v\ sin\theta \tag{2.8}$$

Together with (2.6) we obtain the unicycle model [14], that can be written in matrix form as

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta & 0 \\ sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{2.9}$$

which is the model used in the controller design.

It is important to remark that the unicycle model is non-holonomic, i.e. the number of controllable degree of freedom are less than the total degree of freedom. In fact, there are kinematic constraints on velocity, like the relation between $\dot{x}$ and v_c, or $\dot{y}$ and v_c that makes the two velocities not independent. From a physical point of view it means that some directions of motions are not feasible, like the motion in the direction lateral to the wheel, but the robot is limited to move in straight direction or simply rotate. This means that non-holonomic constraints do not prevent any configuration, so not reduce the configuration space, but limits the generalized velocities of the robot.

It is also possible to notice from the obtained model that the differential-drive robot is a nonlinear dynamical system. It has three state variables, which are assumed to coincide with the system outputs (x,y,θ) and two inputs (v_c,ω), so it is a multiple-input multiple-output (MIMO) system.

REMARK: the presented model represents a first order kinematics model. If we neglect the assumption of no slippage or we consider high velocities, the model should be upgraded including dynamics properties like inertia, mass and slippage. However, in practice, the control of mobile robots using only the unicycle model is very common, due to its simplicity and also because it is quite accurate since robots are mainly used at low speed.

2.1.3 Unicycle model linearization

One of the controller analyzed in this work, i.e. the LQR, requires a linear system, in this section we show how the DDMR model can be linearized with respect to a nominal trajectory to obtain a system of the form (2.34).

We start from the unicycle model in (2.9), which is of the form

$$\dot{\mathbf{x}} = f(x, u, t) \tag{2.10}$$

then we compute the Jacobian matrices of $f(x, u, t)$ with respect to the state vector $\mathbf{x}$ and the control input $\mathbf{u}$ and we evaluate them at the nominal trajectory, in details:

A. We first compute the partial derivatives of $f(x, u, t)$

$$\frac{d\mathbf{f}}{d\mathbf{x}}(\mathbf{x}(t), \mathbf{u}(t)) := \begin{bmatrix} \frac{\partial \dot{x}}{\partial x_1} & \frac{\partial \dot{x}}{\partial x_2} & \frac{\partial \dot{x}}{\partial x_3} \\ \frac{\partial \dot{y}}{\partial x_1} & \frac{\partial \dot{y}}{\partial x_2} & \frac{\partial \dot{y}}{\partial x_3} \\ \frac{\partial \dot{\theta}}{\partial x_1} & \frac{\partial \dot{\theta}}{\partial x_2} & \frac{\partial \dot{\theta}}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 0 & 0 & -v(t)\, sin(\theta(t)) \\ 0 & 0 & v(t)\, cos(\theta(t)) \\ 0 & 0 & 0 \end{bmatrix} \tag{2.11}$$

$$\frac{d\mathbf{f}}{d\mathbf{u}}(\mathbf{x}(t), \mathbf{u}(t)) := \begin{bmatrix} \frac{\partial \dot{x}}{\partial u_1} & \frac{\partial \dot{x}}{\partial u_2} \\ \frac{\partial \dot{y}}{\partial u_1} & \frac{\partial \dot{y}}{\partial u_2} \\ \frac{\partial \dot{\theta}}{\partial u_1} & \frac{\partial \dot{\theta}}{\partial u_2} \end{bmatrix} = \begin{bmatrix} cos(\theta(t)) & 0 \\ sin(\theta(t)) & 0 \\ 0 & 1 \end{bmatrix} \tag{2.12}$$

B. Then we evaluate the Jacobian matrices at the reference state $\mathbf{x}_r(t)$ and input $\mathbf{u}_r(t)$, obtaining the time varying state matrices $\mathbf{A}(t)$ and $\mathbf{B}(t)$, i.e.

$$\mathbf{A}(t) = \begin{bmatrix} 0 & 0 & -v_r(t)\, sin(\theta_r(t)) \\ 0 & 0 & v_r(t)\, cos(\theta_r(t)) \\ 0 & 0 & 0 \end{bmatrix} \tag{2.13}$$

$$\mathbf{B}(t) = \begin{bmatrix} cos(\theta_r(t)) & 0 \\ sin(\theta_r(t)) & 0 \\ 0 & 1 \end{bmatrix} \tag{2.14}$$

2.1.4 Discrete time analysis

The linear system considered in Section 2.1.3 is defined in continuous time, whereas in ROS implementation we have to realize a discrete time controller. For this reason, we have to define a discrete time model of the system.

For this purpose the Euler forward method can be used for discretization [15], which is based on the usage of finite differences for time derivative computation. In particular the derivative of the state can be written as

$$\dot{\mathbf{x}} \cong \frac{\mathbf{x}(k+1) - \mathbf{x}(k)}{T_s} \tag{2.15}$$

where k represents the finite step, and T_s the step size.

Substituting (2.15) in (2.34) we obtain

$$\mathbf{x}(k+1) = \mathbf{A}(k)\ \mathbf{x}(k)\ T_s + \mathbf{B}(t)\ \mathbf{u}(k)\ T_s + \mathbf{x}(k) \tag{2.16}$$

Moreover, it is possible to incorporate T_s in matrices $\mathbf{A}(k)$ and $\mathbf{B}(k)$ having

$$\mathbf{A}_{disc}(k) = \mathbf{A}(k)\ T_s\ , \qquad \mathbf{B}_{disc}(k) = \mathbf{B}(k)\ T_s \tag{2.17}$$

2.1.5 Controllability

Before proceeding in the controller design it is necessary to verify that the system is controllable. Controllability is a property that guarantees the existence of a control function that moves the system from any initial state to any final state in a finite time interval.

Studying the controllability of non-linear systems is quite complex, but for some classes of systems it is possible to derive some general considerations that simplify the controllability study. In particular, the controllability of a drift-less system can be studied using the Rashevsky-Chow theorem as shown in [16] and [17]. In particular the theorem states that if all the constraints of the system are non-holonomic, like in the unicycle case, the system is controllable.

It is also possible to verify the controllability considering the linearization of the system around some equilibrium points or nominal trajectory. In this case, given the Jacobian matices otained from the linearization, we can build the Kalman controllability matrix

$$C = [B, AB, A^2B] \tag{2.18}$$

and verify if the system is fully controllable, i.e. if C is full-rank.

2.2 PID Controller

2.2.1 Introduction to PID Controllers

There are different types of controllers for DDMR, but Proportional-Integral-Derivative controller (PID) is one of the favourite in robot control applications, in particular for trajectory tracking applications [18] [19] [20] [21].

PID controllers are error-based controllers where the control input $u(t)$ of the system is computed evaluating a feedback signal, that in general is the error $e(t)$ between the measured state and the reference one. There are several versions of PID controllers, from the simplest ones, in which is not necessary to know the model of the system, to the more sophisticated and accurate versions, in which are reached better performances introducing the model of the system in the controller design, like for example in the tuning process [22].

Beyond the different versions of PID controllers, all of them are founded on the same base principle of a control action $u(t)$ composed of three main contributions:

- proportional
- integral
- differential

that also give the name to the controller [23].

Proportional action is defined as

$$K_P \, e(t) \tag{2.19}$$

and it is proportional to the feedback error $e(t)$. Using only the proportional contribution, this may lead to some controllability issues. In particular, the controller could be not able to perfectly track a constant reference, with the possibility of introducing a steady-state error, or not being able to reject a constant disturbance.

To cancel the steady state error, integral action can be introduced, defined as

$$K_I \int_0^t e(t) \, dt \tag{2.20}$$

and it is proportional to the integral of the error, over the interval $[0, t]$. This contribution bring to zero the tracking error.

Last, there is the derivative term that is able to reduce the oscillatory behaviour of the response, increasing the damping and obtaining a better stability. The derivative contribution is defined as

$$K_D \frac{d}{dt} e(t) \tag{2.21}$$

and it is proportional to the derivative of the error.

Combining all the three contributions, the total control action is obtained as

$$u(t) = K_P\, e(t) + K_I \int_0^t e(t)\; dt + K_D \frac{d}{dt} e(t) \tag{2.22}$$

2.2.2 PID architecture

The configuration of the PID controller inplented in this work is shown in Fig. 2.2.

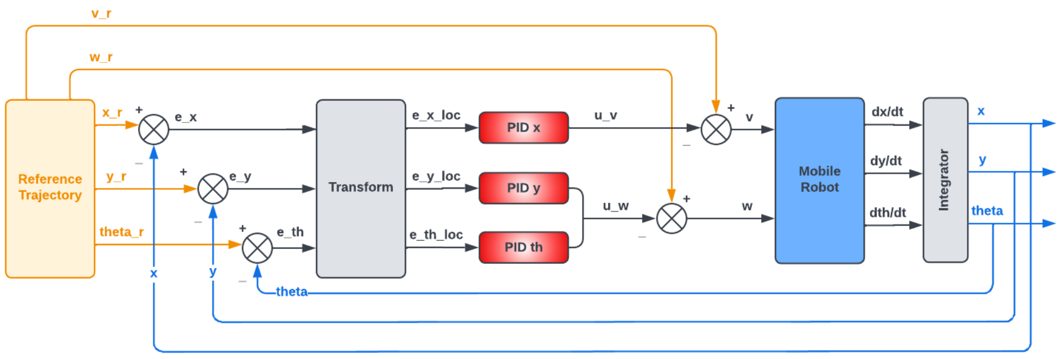


Figure 2.2: Control configuration of PID controller

For non-holonomic systems, usually a feed-forward control is used, in which the control inputs are derived from the reference trajectory [17]. However, in real practice, it is also added a feedback action, realizing a closed-loop control scheme that is robust to errors and disturbances.

As it can be seen in Fig. 2.2, the nominal inputs for the feed-forward action are computed from the reference trajectory, and then are corrected by the outputs of the controller before being sent to the robot.

For the feedback action, the robot actual pose $\mathbf{q}(t) = [x(t), y(t), \theta(t)]^T$ is compared with the reference one $\mathbf{q}_r(t) = [x_r(t), y_r(t), \theta_r(t)]^T$ and consequently, the error vector $\mathbf{e}(t)$ in the global frame is defined as

$$\mathbf{e}(t) = \mathbf{q}_r(t) - \mathbf{q}(t) = [e_x, e_y, e_\theta]^T \tag{2.23}$$

To transform **e**(t) from global frame to the local one we have

$$\mathbf{e}_{loc}(t) = [e_{x_{loc}}, e_{y_{loc}}, e_{\theta_{loc}}]^T = \mathbf{R}^T\mathbf{e} \tag{2.24}$$

where the rotation matrix R is defined as in (2.1).

Once defined the error vector in the local reference frame, the PID control law is chosen as shown in [21] and [11]

$$\begin{cases} u_v = K_{p_x} e_{x_{loc}} + K_{i_x} \int e_{x_{loc}} dt + K_{d_x} \frac{d}{dt} e_{x_{loc}} \\ u_\omega = K_{p_y} e_{y_{loc}} + K_{i_y} \int e_{y_{loc}} dt + K_{d_y} \frac{d}{dt} e_{y_{loc}} + \\ \qquad + K_{p_\theta} e_{\theta_{loc}} + K_{i_\theta} \int e_{\theta_{loc}} dt + K_{d_\theta} \frac{d}{dt} e_{\theta_{loc}} \end{cases} \tag{2.25}$$

With this choice, the control input related to linear velocity u_v depends only by the error on x coordinate. The reason is that in DDMR the linear velocity has only a component along x axis of robot's frame, due to the non-holonomic constraints of the robot. On the other hand, the control input in terms of angular velocity u_ω, is related to the error on y axis and orientation θ, because angular velocity depends by the orientation angle, but also by the component perpendicular to linear velocity, that is the y component of position in robot's frame.

This control action represents the correction to the nominal input of the feed-forward action ($v_r(t)$, $\omega_r(t)$) and for this reason it is subtracted from it. Then the obtained control is given as input to the robot, as shown in Fig. 2.2.

This control law is defined in continuous time, while in the ROS node it is implemented in discrete time. For this reason, the PID control law is written in discrete form [24]: the error vector **e** is computed at each step k, obtaining

$$\mathbf{e}_k = [e_{x_k}, e_{y_k}, e_{\theta_k}]^T \tag{2.26}$$

while the integral of error is approximated as summation of previous errors, i.e.

$$\int_0^t \mathbf{e}(t)dt \cong T_s \sum_{j=0}^{k} \mathbf{e}_k \tag{2.27}$$

and the differential error is defined as difference between previous and actual error, i.e.

$$\frac{d}{dt}\mathbf{e}(t) \cong \frac{\mathbf{e}_k - \mathbf{e}_{k-1}}{T_s} \tag{2.28}$$

With these considerations, the actual PID control law implemented in the ROS node is defined as:

$$\begin{cases} u_{v_k} = K_{p_x} e_{x_k} + K_{i_x} \sum_{j=0}^{k} e_{x_k} + K_{d_x}(e_{x_k} - e_{x_{k-1}}) \\ u_{\omega_k} = K_{p_y} e_{y_k} + K_{i_y} \sum_{j=0}^{k} e_{y_k} + K_{d_y}(e_{y_k} - e_{y_{k-1}}) + \\ + K_{p_\theta} e_{\theta_k} + K_{i_\theta} \sum_{j=0}^{k} e_{\theta_k} + K_{d_\theta}(e_{\theta_k} - e_{\theta_{k-1}}) \end{cases} \tag{2.29}$$

where T_s is incorporated in the PID coefficients.

One of the problems of PID controllers is the integral windup, that occurs when there is a large change in reference input, that cause a large error, which is accumulated in the integral contribution [25]. To have better performances, it is possible to implement an anti-windup mechanism [26] [23], that works saturating the integrator when the integral term reaches a certain threshold and the integrator output has the same sign of PID input. The integration restarts when integrator output and PID input have opposite sign.

We want to remark that the mechanical constraints on the max v and ω reachable by the robot are ensured by a saturation function after the PID block.

2.3 LQR Controller

2.3.1 Introduction to LQR Controllers

Together with PID controllers, Linear Quadratic Regulator (LQR) is one of the technique most used for trajectory tracking of DDMR [12].

LQR controllers are developed in the field of optimal control, that is a branch of mathematical optimization aimed to find a control for a dynamical system optimizing an objective function [27]. In particular, LQR controllers are applied to linear systems, minimizing a quadratic objective function.

LQR problem can be formulated in a finite horizon, considering as final time a finite value, or can be defined in an infinite horizon if the final time tends to infinity. Considering the finite horizon case, the cost function J to minimize is expressed as

$$J = \phi(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} L(\mathbf{x}(t), \mathbf{u}(t), t)dt \tag{2.30}$$

where t_0 is the initial time instant and t_f is the final time. The first function $\phi(\mathbf{x}(t_f), t_f)$ represents the contribution of the final state $\mathbf{x} \in \mathbb{R}^n$ of the system at time t_f, so it is a weight related to the error between the initial and final state. Its is expressed as a quadratic function of the final state $\mathbf{x}(t_f)$

$$\phi(\mathbf{x}(t_f), t_f) = \frac{1}{2}\mathbf{x}^T(t_f)\,\mathbf{F}\,\mathbf{x}(t_f) \tag{2.31}$$

where $\mathbf{F} \in \mathbb{R}^{n \times n}$, $\mathbf{F} \succ 0$ is the weight matrix related to the final state, and it is also a design parameter.

The function $L(\mathbf{x}(t), \mathbf{u}(t), t)$ is related to the entire control interval $[t_0, t_f]$ and it takes into account both the state $\mathbf{x}(t)$ and the control input $\mathbf{u}(t) \in \mathbb{R}^m$. It can be written as a quadratic function of the form

$$L(\mathbf{x}(t), \mathbf{u}(t), t) = \frac{1}{2}\mathbf{x}^T(t)\,\mathbf{Q}(t)\,\mathbf{x}(t) + \frac{1}{2}\mathbf{u}^T(t)\,\mathbf{R}(t)\,\mathbf{u}(t) \tag{2.32}$$

where $\mathbf{Q} \in \mathbb{R}^{n \times n}, Q \succeq 0$ is the weight matrix related to the state $\mathbf{x}(t)$, while $\mathbf{R} \in \mathbb{R}^{m \times m}, R \succ 0$ is the one related to the input $\mathbf{u}(t)$.

Correspondingly, the cost function can be explicitly written as

$$J = \frac{1}{2}\mathbf{x}^T(t_f)\,\mathbf{F}\,\mathbf{x}(t_f) + \int_{t_0}^{t_f} \frac{1}{2}\mathbf{x}^T(t)\,\mathbf{Q}(t)\,\mathbf{x}(t) + \frac{1}{2}\mathbf{u}^T(t)\,\mathbf{R}(t)\,\mathbf{u}(t)dt \tag{2.33}$$

A practical interpretation of LQR approach is following: we want to chose the control action $\mathbf{u}(t)$ that minimize the cost function (2.33), leading the final state

$\mathbf{x}(t_f)$ to zero and enforcing the weighted energy due to the state and the control input to be as low as possible for the entire control time. Matrices $\mathbf{F}, \mathbf{Q}(t)$ and $\mathbf{R}(t)$ define the weight of each contribution and represent the design parameters of the controller, to be tuned for obtaining the desired performances. $\mathbf{Q}(t)$ and $\mathbf{R}(t)$ can be time-varying, but in general are chosen fixed values.

In many cases, the cost function is defined with zero terminal cost, so considering a null matrix for $\mathbf{F}$. In this case, the only design parameters of the controller are matrices $\mathbf{Q}$ and $\mathbf{R}$.

The system state $\mathbf{x}(t)$ and input $\mathbf{u}(t)$ enter the problem considering the state space representation of the system, i.e.

$$\dot{\mathbf{x}} = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \tag{2.34}$$

Then, the LQR design is achieved through the solution of an optimization problem formulated as

$$\begin{gathered}\text{Find } \mathbf{u}(t) \text{ that minimizes: (2.33) in } t \in [t_0, t_f] \\ \text{subject to: (2.34),} \\ \mathbf{x}(t_0) = \mathbf{x}_0\end{gathered}$$

2.3.2 Solution of LQR problem

The LQR optimization problem in section 2.3.1 can be solved using Lagrange multipliers method, as shown in [28]. It is a technique for finding the local minima and maxima of a function with equality constraints.

In this problem, the optimization variable is the input $\mathbf{u}(t) : [t_0, t_f] \to \mathbb{R}^m$ and there are an infinite number of equality constraint, one for each $t \in [t_0, t_f]$, represented by the state space equation of the system in (2.34).

The Lagrange multiplier function is defined as $\lambda(t) : [t_0, t_f] \to \mathbb{R}^3$. It can be used to define the Lagrangian function $\mathcal{L}$ as

$$\mathcal{L}(\mathbf{x}, \lambda) = J + \int_{t_0}^{t_f} \lambda(\tau)^T (A\mathbf{x}(\tau) + B\mathbf{u}(\tau) - \dot{\mathbf{x}}(\tau)) d\tau \tag{2.35}$$

The stationary points of $\mathcal{L}$ can be computed considering the points where its gradient is null, i.e.

$$\nabla_{u(t)}\mathcal{L} = \mathbf{R}\ \mathbf{u}(t) + \mathbf{B}^T\ \lambda(t) = 0 \tag{2.36}$$

$$\nabla_{x(t)}\mathcal{L} = \mathbf{Q}\ \mathbf{x}(t) + \mathbf{A}^T\ \lambda(t) + \dot{\lambda}(t) = 0 \tag{2.37}$$

from (2.36) we get

$$\mathbf{u}(t) = -\mathbf{R}^{-1}\mathbf{B}^T\lambda(t) \tag{2.38}$$

while from (2.37) we get

$$\dot{\lambda}(t) = -\mathbf{A}^T - \mathbf{Q}\mathbf{x}(t) \tag{2.39}$$

(2.34) and (2.39) are called co-state equations.

The control input $\mathbf{u}(t)$ in (2.38) is the one that minimizes the performance index and using it we can write the co-state equations in matrix form, called Hemiltonian matrix, as

$$\frac{d}{dt}\begin{bmatrix}\mathbf{x}(t)\\ \lambda(t)\end{bmatrix} = \begin{bmatrix}\mathbf{A} & -\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\\ -\mathbf{Q} & -\mathbf{A}^T\end{bmatrix}\begin{bmatrix}\mathbf{x}(t)\\ \lambda(t)\end{bmatrix} \tag{2.40}$$

with initial condition

$$\mathbf{x}(t_0) = \mathbf{x}_0 \tag{2.41}$$

The relationship between $\mathbf{x}(t)$ and $\lambda(t)$ can be expressed defining a value function $K(t)$, for which

$$\lambda(t) = K(t)\mathbf{x}(t) \tag{2.42}$$

such that

$$-\dot{K} = \mathbf{A}^T K + K\mathbf{A} - K\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T K + \mathbf{Q} \tag{2.43}$$

(2.43) is satisfied if a $K(t)$ can be found such that

$$\dot{K} = K\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T K - K\mathbf{A} - \mathbf{A}^T K - \mathbf{Q} \tag{2.44}$$

and it is called algebraic Riccati equation. This equation can be solved by backwards numerical integration, and it is independent of initial conditions $\mathbf{x}_0$. The solution $K(t)$ is a positive semi-definite matrix and can be used to design the optimal control law as a state feedback, i.e.

$$\mathbf{u}(t) = -\mathbf{R}^{-1}\mathbf{B}(t)^T K(t)\mathbf{x}(t) \tag{2.45}$$

If we define the LQR gain $K_{lqr}(t)$ as

$$K_{lqr}(t) = -\mathbf{R}^{-1}\mathbf{B}(t)^T K(t) \tag{2.46}$$

then (2.45) becomes

$$\mathbf{u}(t) = K_{lqr}(t)\mathbf{x}(t) \tag{2.47}$$

This is the optimal control law used to construct the feedback system.

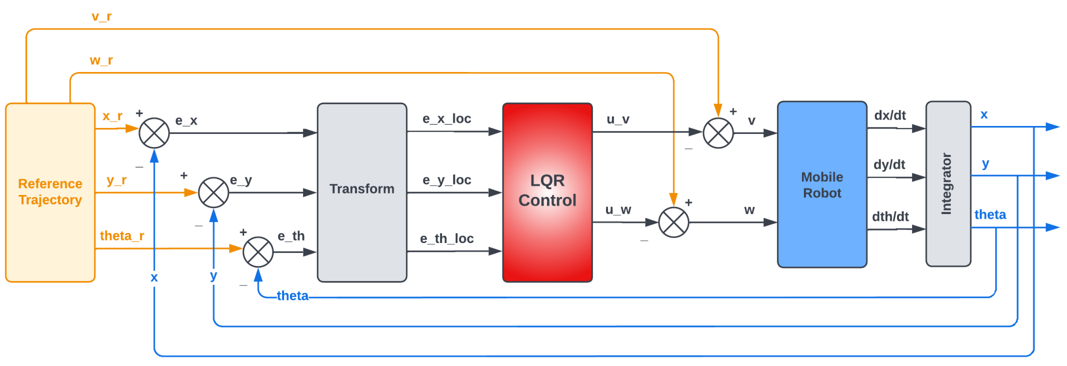


Figure 2.3: Control configuration of LQR controller

2.3.3 LQR architecture

The block scheme of the LQR controller is depicted in Fig. 2.3.

As shown in Fig. 2.3, the LQR has beed designed for the error dynamics. Indeed, the error between the reference robot's pose and the actual one is considered as state, i.e.

$$\mathbf{e}(k) = \begin{bmatrix} e_x(k) \\ e_y(k) \\ e_\theta(k) \end{bmatrix} := [\mathbf{x}_r(k) - \mathbf{x}(k)] = \begin{bmatrix} x(k) - x_r(k) \\ y(k) - y_r(k) \\ \theta(k) - \theta_r(k) \end{bmatrix} \tag{2.48}$$

In this way, rather than the system state, the controller allows to steer to zero the tracking error.

As shown for the PID controller, also in this case the obtained control input is subtracted from the nominal inputs $\mathbf{u}_r$ generated from reference trajectory. In fact, the control action represents a correction around the feed-forward action. In conclusion, the optimal feedback control law becomes

$$\mathbf{u}(k) = -K_{lqr}(k)[\mathbf{x}_r(k) - \mathbf{x}(k)] + \mathbf{u}_r(k) \tag{2.49}$$

This is translated in the block scheme in Fig. 2.3, with the actual robot pose that is fed back and compared to the reference pose to compute the state error $\mathbf{e}(k)$ in (2.48).

Chapter 3

ROS manual

This chapter presents an overview about the tools used to accomplish this project thesis. The large part of the work is based on ROS, but it is also integrated with other software, i.e. Gazebo for the simulation part and Matlab for the control algorithm development.

This section is conceived as a manual for a reader that is approaching ROS for the first time, offering a presentation of the most important features of this environment and a practical guide for the initial setup of a ROS project, and the integration with Gazebo and Matlab.

This manual does not have the presumption to replace the official ROS manual, that can be found at [29] or other valid guides like [30] or [31], because also this work contains code snippets from The ROS Wiki at [29], available under Creative Commons Attribution 3.0.

In the first part of this chapter, an introduction to ROS is presented, showing its strong points, then the main elements of ROS architecture are analyzed. After that, the Gazebo simulator and its components are introduced. The next section explains the basic steps to setup ROS, based on the experience gained with this work. Later, the integration between ROS and Gazebo is discussed and the chapter ends with a presentation of integration between ROS and Matlab, based on the use of ROS Toolbox.

3.1 ROS introduction

ROS stands for Robot Operating System and it is a software framework for robotics. It is similar to an operating system, so provides similar services, i.e. error handling, processes management and communication, file system and user interface, but it is not a standard one, because it can not run stand-alone, but requires an existing operating system, like Linux. In addiction to operating system services, it provides also other functionalities and libraries, that are specific for robotic development, suitable for algorithm design, test, simulation, data saving and debugging.

ROS is not the only robotic frameworks, like shown in [32]; however, nowadays ROS can be considered the standard for robot platforms. Below are listed some of the main reasons that lead to this result. A key element is the open-source nature of the ROS project, that allow it to have widespread collaborations with universities and research institutes and so a lot of contributes to its development. For this reason, it can be considered a community-driven project, supported worldwide. Another important feature is the code modularity, reusability and sharing of ROS projects. In this sense, a wide repository of projects has been developed, that are disposable for all the ROS uses. This allows to reduce developing time and encourage collaboration. Another strength of ROS is the language independence, i.e. each code part can be designed using different supported languages and the most used are C++ and Python, for which is ensured the support to ROS libraries. ROS provides also a hardware abstraction layer, allowing developers to build robotics applications without considering the underlying hardware.

A ROS project is organizes in packages, that group piece of code and processes with same functionalities. Packages can be published to public repositories in order to guarantee their distribution. Sets of ROS packages are organized in distributions, that are released with progressive versions. The last release of ROS, considering actual year 2022, is called "Noetic" and it will be supported until 2025.

There are also two different versions of ROS, called "ROS" (or "ROS1") and "ROS2". ROS2 is the newer version, developed from the first one, adding the required modifications to improve the environment [33]. ROS2 was developed from scratch and not modifying ROS1, to avoid many changes and making it unstable. ROS2 is being developed to satisfy industrial requirements, like real-time, safety, certification and security. In this thesis we have been focused on ROS1, since it is still the most used until now, but it is not an useless job, since ROS2 is backward compatible with ROS1 and we could easily make some ROS2 nodes communicated with old ROS1 nodes.

3.2 ROS architecture

ROS represents a middleware framework for process communication, allowing exchange of data between them.

It is structured as a distributed framework, since the processes, called nodes, can be designed and executed individually and can run also in different machines, even far from each other.

The ROS communication network is called "graph" as described in [34], and its main elements are:

- Nodes: the processes that compose the graph. Robot projects are based on many nodes with specific functions, like path planning, sensor data acquisition or motors control.
- Master: main node that manages the communication between processes. It enables a node to locate another one, and then they communicate peer-to-peer. It acts like a DNS server in a computer network, providing look-up information for the other nodes. Like computer networks, also in ROS protocols are used to manage the communication, such as the TCPROS transport layer, based on TCP/IP sockets.
- Parameter server: a shared dictionary, provided by the master. It is used by the nodes to store parameters at run-time.
- Messages: data structures that are exchanged between nodes, allowing their communication. There are different types of messages, and some of them are designed for specific sensors, like radars or cameras. It is also possible to define a custom messages, with a custom data structure.
- Topics: busses over which nodes exchange messages. Each topic has a name and it is associated to a specific type of message, so only the selected type can be sent over the topic. A node that send messages on a topic is called "publisher", while one that receives messages is called "subscriber". Both of them needs to be registered to the master before sending or receiving messages. Topics are designed for a communication paradigm based on many-to-many and one-direction transport.
- Services: other communication channels as well as topics, but they are designed for request-reply paradigm, that is commonly used in a distributed system. A service is defined by a pair of messages, i.e. one for the request and one for the reply, and it is implemented in a certain node, that receive the request by a client node, and send back the reply.
- Bag files: files used to save ROS messages. They can be used to store data from sensors or commands, that can be played back, to simulate a certain input for example.

To better understand the ROS graph architecture, a simple scheme is reported in Fig. 3.1 and Fig. 3.2

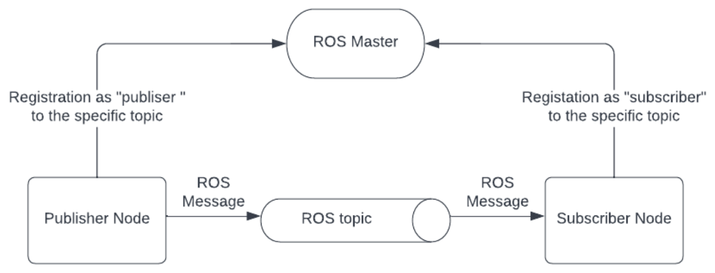

ROS Master
Registration as "publiser " to the specific topic
Registation as "subscriber" to the specific topic
Publisher Node
ROS Message
ROS topic
ROS Message
Subscriber Node

Figure 3.1: ROS graph structure

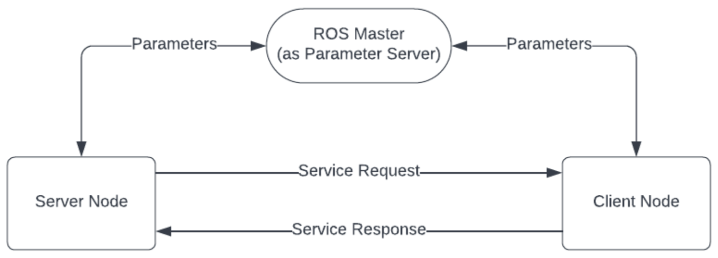


Figure 3.2: ROS graph structure (for server)

ROS presents a specific graphical plugin for visualizing the computation graph, called *rqt_graph*. It shows all the node in execution, the connections with topics, and services, in form of a graph. Considering a simple graph with a publisher and a subscriber on two topics, the *rqt_graph* output is reported in Fig. 3.3.

3.3 Gazebo introduction

Simulators are very important to design and test a robotic system. They allow to design a robot, model the sensor behaviour, and test algorithms.

Gazebo is one of the most used simulators in robotics, especially because it is well integrated with ROS. In fact it is maintained by the "Open Robotics" corporation, that is the same responsible for ROS, but at the same time, Gazebo is an

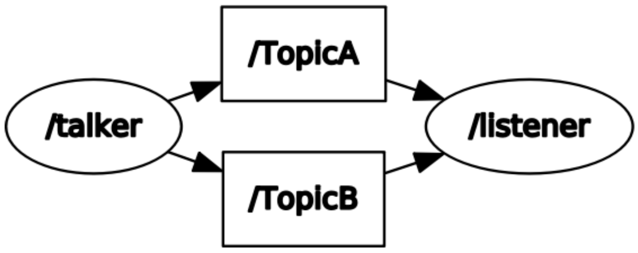


Figure 3.3: Example of *rqt_graph* output

independent project, that exists also without ROS.

As described in [35], Gazebo is an open source 3D simulator that reproduces the model dynamics and its interaction with environment and obstacles using a physical engine. Gazebo has the option to chose different engines, but the most used is the Open Dynamic Engine (ODE).

To define a robot in Gazebo, two types of file can be used:

- Unified Robotic Description Format (URDF): is an XML format, that describes the element of the robot. It is limited to specify only the kinematics and dynamics of the robot, and it is not able to specify the pose, friction or other properties.
- Simulation Description Format (SDF): is a specific format for Gazebo, created to overcome URDF limitations. It allows to completely describe a robot, but can be used also to describe a world environment. SDF is the main file format used in Gazebo, and also URDF files are automatically converted to SDF to be used.

Another component of Gazebo are the plugins, that can be considered as shared libraries used to connect the simulator with ROS. In general, plugins are defined using a URDF file. Plugins can be used to access the simulated model and extract data about the physics of the robot model or sensors, and share them with ROS, other than converting ROS messages into physical commands in the simulation environment.

There are also many libraries of robots and environments that can be integrated in Gazebo that allow to reduce the developing effort, using resources already working and tested.

3.4 ROS setup

ROS is not a stand-alone operating system, but needs to run on an existing one. It is possible to install ROS on different operating systems, but the most common choice is to use Ubuntu, because it is the only one, together with Debian, to be officially supported. ROS can run on a full installation of Ubuntu, but also on a virtual machine. For this work we have used a full installation of Ubuntu 20.04, that is the last LTS (long-term support) version in the period of this work (2022).

After installation, it is possible to use ROS commands in bash terminals, but to do so we have to run a specific script in each terminal, as highlighted in [36]. To avoid this, we can add this command in the bash source file ".bashrc", in this way the script is automatically sourced at every shell launched.

To create a first ROS project, we defined a workspace where all the ROS files are stored. The most used approach is to use a catkin workspace, represented by a folder, in general called "catkin_ws", created in the home directory. This folder is not already present, but needs to be created manually. Catkin is the build system for ROS, as indicated in [37], that has the function of compiling code to create an executable file. Catkin uses cmake to generate build files, and each package has a CMakeLists.txt, where are configured the files to build. To build the files in the workspace, the bash command *catkin_make* is used inside its folder. At the first execution, this command created also all the folders and files necessary for the workspace.

Catkin workspace is composed by four main folders, called *spaces*, that are:

- src: source space, which contains the source code of packages;
- build: build space, that stores cache information and intermediate files of the catkin build process;
- devel: development space, containing the built targets before the installation;
- install: install space, that is the space where the built targets are installed.

It is important to remark that the development phase is fully performed in the *src* space.

At this point we can start a new project, creating a ROS package. Packages are folders that contains all the files related to a project, and they are stored in the *src* folder. To create a package there is a specific bash command, as shown in [38]. In the command, the package name and the list of dependencies, like Python or specific ROS messages, are included.

Each package contains some folders created by default:

- include: where are placed additional files to include during the building
- src: contains the C++ source files of nodes

In addition other folders are added manually in general, for a better organization of the package, i.e.

- launch: to store launch files, used to run multiple nodes
- scripts: to store Python source files of nodes
- bagfiles: to store topics data, saved in bag form

After this overview of ROS file-system it is possible to analyze the usage and design of ROS nodes, as shown in [39]. A node contained in a certain package can be run using the command

```
rosrun [package_name] [node_name]
```

Before running any node, we have to run the ROS-based system, that is a collection of nodes that perform the basic functions of the ROS environment. This is done using the bash command

```
roscore
```

which will start-up the ROS master, the parameter server and a logging node, called "rosout".

If a project is composed by many nodes, there is a faster way to start all the processes at same time, that is the usage of a roslaunch file, using the command

```
roslaunch [package] [filename.launch]
```

A launch file is similar to an xml file, and an example is reported in appendix A.0.1, where it is shown a launch file used in this work for the simulation of sinusoidal trajectory, using PID controller. It is possible to notice that in a launch file we can call also a service, for example to reset the Gazebo environment before each simulation. We can also start the Rosbag data saving, allowing to do it at the same time of a node launch. We can specify the time interval to save and select the topics whose message are stored.

When we start a node in a launch file, there are also some parameters to set, and one of them is

```
output="screen"
```

that is used when a node prints something on the terminal.

One of the key feature of ROS is the communication between nodes. This is done exchanging messages through the topics. Messages are specific data structures defined in ROS, stored in *.msg* files. Standard messages are able to represent all the most common data used in robotics. For example $Vector3$ is able to represent a vector, containing the data of the three vector components, $Twist$ to represent a velocity, with linear and angular components, $Twist\_stamped$ in which it is added also a header with a timestamp. In appendix A.0.2, the structure of some commonly used messages is shown as an example.

As anticipated before, nodes are the main core of a ROS project. There are two main approaches to design a node: using C++ or Python. In both cases, a client library is provided that enables the programmers to interface with ROS. For C++, the library is called *roscpp*, and it is designed to be an high-performance library, while for Python is *rospy*, that favors implementation speed over runtime performance. In both cases the node structure is characterized by:

- node initialization, in which the node name is defined
- definition of node type, e.g. publisher or subscriber. We want to remark that the same node can be also publisher & subscriber at the same time. Moreover, for each node we also need to define the topic on which messages are exchanged.

Specific commands are defined in [40] for C++ and [41] for Python. The only difference between C++ and Python is that in the first approach the initialization of the node is performed by a handle, in addition to the initialization definition, so there is an additional command to consider.

To build the node there are two different approaches between C++ and Python. A C++ code needs to be build before execution, since it is a compiled language. ROS code is build using catkin, and the configuration file is the "CMakeLists.txt" contained in the package folder. This file should be configured in the section "Build", with the functions

```
add_executables(<exec_name> <src_path>)
target_link_libraries(<exec_name \${catkin_LIBRARIES}>)
```

where the executable name, the folder path, and the target libraries are specified. In case of a node written with multiple files, in the first function, in $< src\_path >$ all the files .cpp are written, while the second function is written in the same way. It is not needed to add the .h files, that are already included in .cpp files.

Then the code is built using the command *catkin_make* in catkin workspace.

Python, instead, is an interpreted language, so it is not needed to generate an executable, but also in this case we have to run the *catkin_make* command to make sure that the autogenerated Python code for messages and service is created. In this case it is not needed to modify the "CMakeLists.txt" file.

Every Python node should start with the line

```
#!/usr/bin/env python
```

that makes sure that the script is executed as a Python script.

3.5 Integration between ROS and Gazebo

Gazebo is a stand-alone simulator, but can be integrated with ROS using a set of ROS packages, called *gazebo_ros_pkgs*. They provide the interface to simulate a robot in Gazebo using ROS messages and services.

A simulation setting consists of two main parts: the robot model, that can be designed manually or imported from one of the packages available on repositories, and the environment, with possible obstacles.

In this work, for example, a model called "Waffle", from the TurtleBot3 package was used. It is shown in Fig. 3.4.

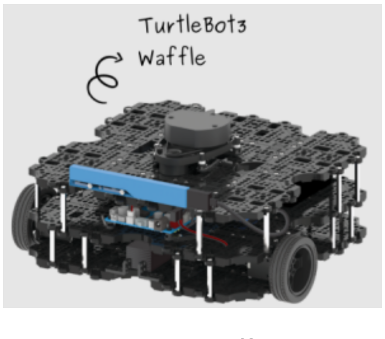


Figure 3.4: TurtleBot waffle - Image from [42]

TurtleBot is a robot platform with an open-source software, that is one of the most used platforms for ROS among developers and students [10]. It is an hardware platform but for simulation purposes it is provided also the Gazebo model, that allows to make realistic tests before using the real robot. The choice of this model derives from its large usage and extensive documentation, that allow to be more supported during the development.

There are different versions of TurtleBot, but it is chosen TurtleBot3 because is one of the last developed and more complete packages.

The Waffle model can be imported in Gazebo, setting an environment variable with a bash command and then the simulator is launched with a launch file, also provided in the chosen package, loading the world with the spawned model. The used bash command is

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

For the environment, the *Empty_world* model is considered, i.e. a basic scenario containing only an infinite flat ground, that allows to simulate the interaction with robot's wheels.

In Fig. 3.5 is reported a view of the Gazebo environment with robot model.

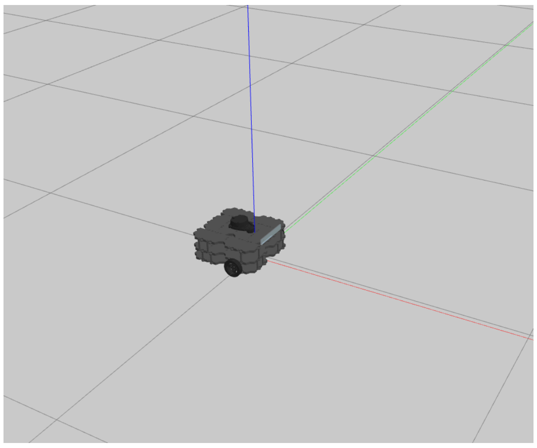


Figure 3.5: TurtleBot3 Waffle in Gazebo Empty_world

In the launch file there are different options to set the initial conditions for the robot spawning, like the starting position and orientation. Another important option is to set

```
<arg name="use_sim_time" value="true"/>
```

that allows to use as reference time for ROS the Gazebo simulation time, in this way the two environment are synchronized.

Once the simulation environment is set, a control algorithm is required to guarantee that the robot follows the desired path. This is done using the ROS integration with Gazebo. In fact Gazebo can communicate with ROS acting like a normal node, with some plug-ins that allow to subscribe to certain nodes, for receiving commands, or to publish to other nodes, for sending information about the simulation state, like the output of sensors applied to the robot. Hence, the communication between Gazebo and ROS occurs exchanging messages on topics between nodes, like in any other ROS application.

For what concern the Waffle robot for example, it can be controlled publishing the velocity command on the *cmd_vel* topic. This command is defined using the ROS message *geometry_msgs/Twist*, that contains the linear and angular component, as shown in appendix A.0.2. This command input is very common to a lot of mobile robots.

A problem with this input can be the absence of an header in the message type *Twist*, so the lack of a time reference in the data. This can be a problem for data visualization, since after being saved in a rosbag file, the only time reference is the header, and in this case it is not present.

A possible solution can be the publication of the same command input on two topics simultaneously, i.e. *cmd_vel*, already described, and a new one, *cmd_vel_stamped*, in which the ROS message *geometry_msgs/TwistStamped* is published. This message is a variation of the normal *Twist* in which an header with a time stamp is added, containing the time reference of the message creation. In this way *cmd_vel* is still the input of the robot, while *cmd_vel_stamped* is used only for data visualization.

In addition, the Waffle robot publishes its state on the topic *odom* in which sends its pose and velocity, obtained from its sensors. This information is insert in the message *nav_msgs/Odometry*, that can be find in appendix A.0.2.

Hence, from ROS topic *odom* it is possible to read the robot's pose, while on topic *cmd_vel* it is possible to send the command input. These topics can be used by a control algorithm implemented in a ROS node to make the robot move.

An example of simulation view in Gazebo is shown in Fig. 3.6 where we have tried to give the movement idea superimposing different simulation frames.

3.6 Integration between ROS and Matlab

Integration between ROS and Matlab is achieved using a specific toolbox, i.e. the *ROS Toolbox*, in which specific functions to create a publisher or subscriber node

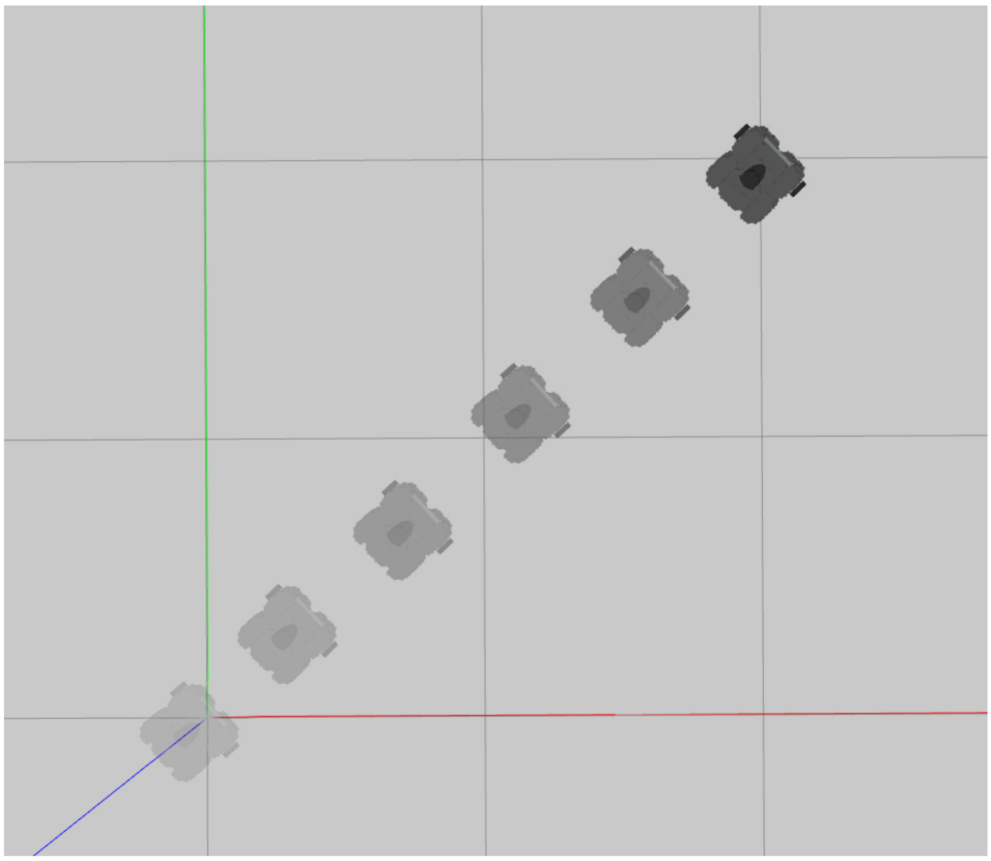


Figure 3.6: Waffle in movement along straight trajectory

inside a Matlab script are developed. In this way, Matlab can be connected to a ROS network, and explore available topics and services, allowing to send commands and receive data from any node on the ROS network. A scheme of the integration between Matlab and ROS is reported in Fig. 3.7

ROS Toolbox was included for the first time on the R2015A Matlab's release, and it has been a great innovation, because before the most common way of connecting ROS with Matlab, was using a bridge developed in Java, as clarified by [9]. ROS Toolbox is suitable also for integration between ROS and Simulink, since the same functions developed for Matlab are also available as Simulink blocks.

The advantage of integrating Matlab and ROS is to have the possibility to develop a control algorithm using all the power-full Matlab tools and integrate them with the versatility of ROS. ROS Toolbox supports also C++ code generation using the Matlab Coder, in this way the developed controller can be used as a stand-alone ROS node.

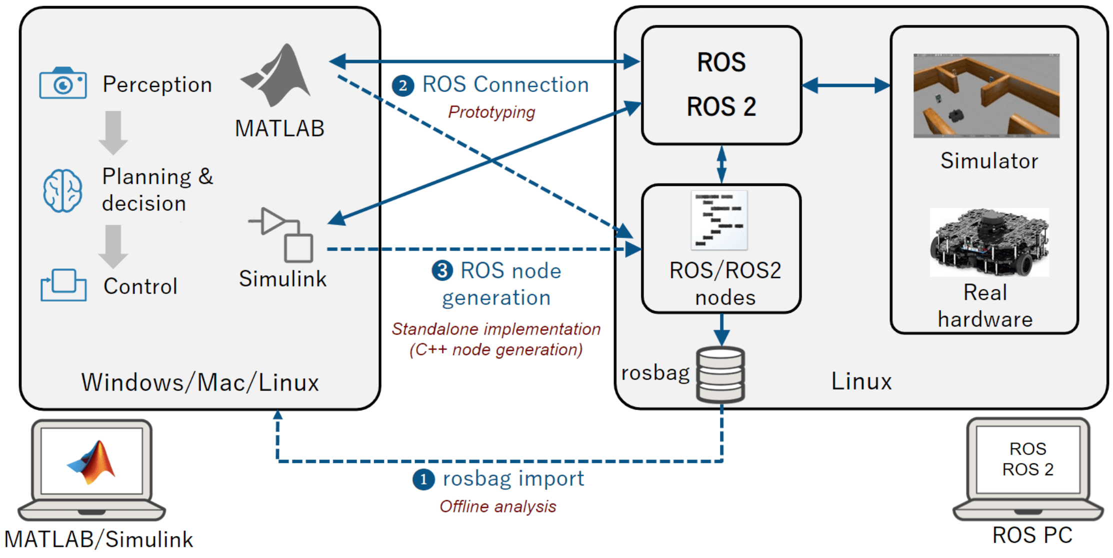


Figure 3.7: Integration between Matlab and ROS. Image from [43]

The toolbox also supports all ROS messages, that are included as a library. ROS messages are stored in Matlab as message structures or objects, but structure is the recommended format since has better performances and is the only supported format in Matlab Coder.

REMARK: Installing Matlab on Ubuntu can be a bit tricky, since after simple installation there can be some settings to fix. Here are some tips: during installation, select "create Symbolic link". In this way,the link to the executable is created, allowing Matlab's launch from terminal.
Then Matlab is installed, in general, at location (for version R2022a)

```
/usr/local/MATLAB/R2022a
```

If we want to install other toolbox or add-ons after the first installation, we have to modify permissions of this folder, to allow Matlab to change its content. This is done with the bash command

```
sudo chown -R \$LOGNAME: /usr/local/MATLAB/R2022a
```

where the last part is related to the Matlab installation path.

In some cases, it may happen that, after installation, the Matlab icon is not present in the Ubuntu app launcher. This can be solved adding, at path

```
$/usr/share/applications/$
```

a file that can be called, for version 2022a, "matlab_r2022a.desktop", which content is reported in A.0.3. The section related to "Exec" is aimed to solve a possible error relative to graphic drivers.

3.7 C++ code generation with Matlab Coder

To generate a code from Matlab script, we must respect some requirements. One of them, as cited before, is the usage of only structure messages, because objects are not supported. Another requirement to remark is that it is possible to export only a function, so the code needs to be written in this form, or another constraint is the ban on the use of global variables, because they are not optimized. Information on compatibility to Matlab Coder is reported in the documentation of each function.

Then the function can be exported in C++ selecting some build options. Matlab Coder support directly the code generation for ROS, since it is possible to select as target hardware board the ROS option. This option presents some restrictions, since the function to convert should not have any input or output, but should contain the publisher or subscriber definition to communicate with ROS network. In some cases, this can be limiting, since a common approach is to export only the controller function, that has some input and output, and so the ROS option can not be selected as hardware. In this case, it is still possible to use as option "MATLAB Host Computer", that allows the target function to have inputs and outputs.

The Matlab Code generator options used for this work are shown in Fig. 3.8

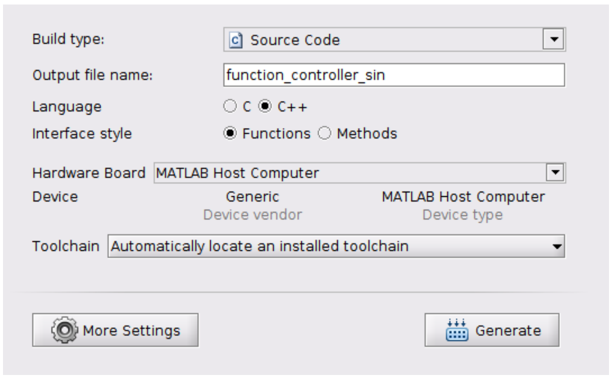

Build type: Source Code
Output file name: function_controller_sin
Language: C [✓ C++]
Interface style: [✓ Functions] Methods
Hardware Board: MATLAB Host Computer
Device: Generic (Device vendor), MATLAB Host Computer (Device type)
Toolchain: Automatically locate an installed toolchain

Figure 3.8: Matlab Coder settings

3.8 Data visualization

Data from ROS topics can be saved using *rosbag* files. ROS offers several tools for data visualization and debugging of these files. The most used are *RViz*, that allow to visualize a 3D data of messages of specific sensors, like cameras, and *rqt* that is more suitable for visualization of single-dimension messages, plotted over time.

Rosbag files can be also imported in Matlab using specific functions of ROS Toolbox. After importation, it is possible to filter and extract message data, that can be used as normal Matlab data. This allows to use all the Matlab functionalities for data visualization, like the usage of plot functions.

3.9 Controller node structure

As shown in the previous sections, there are four main alternatives to write a ROS control node, i.e. using C++, Python, Matlab or C++ from Matlab, and the goal of this work is to compare these approaches to underline the different performances. To do that, the control algorithm is implemented with the same structure in the four cases, to have a fair comparison.

The basic structure of all the algorithm is shown in Fig. 3.9

- Initialization part, in which the Gazebo environment is reset, making a call to a specific service. In this way the robot re-spawn with the starting pose and the simulation time is reset to zero. This is done to have the same initial conditions at each simulation.

- Start of Rosbag data saving, that allow to store in a file the messages published to the topics of interest.

- Subscription to the *odom* topic, to receive messages on robot's pose, and registration as publisher on *cmd_vel* topic, to send the computed control input. With this setting, each time a message is published on *odom* topic, a callback function is activated. In this one the variables of actual robot's position and orientation are updated, and used later in the control command computation. Odom messages are published with a rate of 30 Hz, that depends by the robot model definition in Gazebo.

- Control action computation, which is implemented in a loop running at 20 Hz, that compute the control action, according to the PID or LQR algorithm. Then the command is sent to the robot's input, publishing on *cmd_vel* topic.

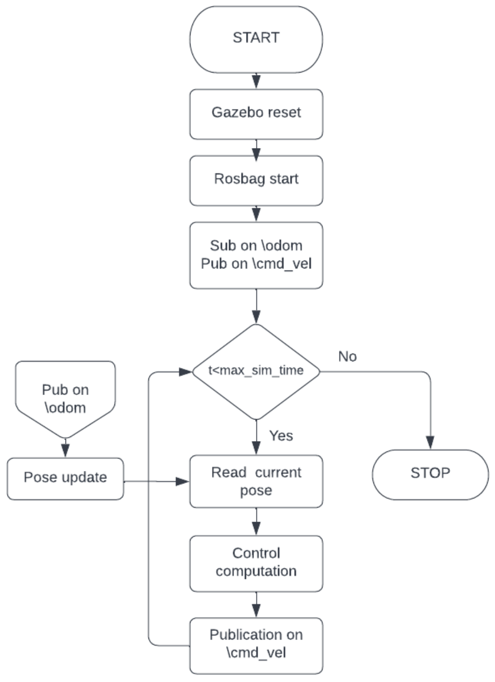


Figure 3.9: Control node flowchart

The simulation setup can be represented also from the point of view of the ROS graph. The output of the *rqt_graph* command is shown in Fig. 3.10

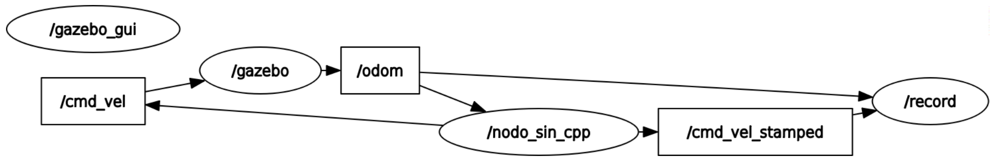


Figure 3.10: ROS graph for the simulation setup

there are three main nodes:

- gazebo: is the node of the simulator, that receive as input the messages on topic *cmd_vel* and has as output the robot odometry state on topic *odom*;
- nodo_sin_cpp: is the controller node, considering the case of sinusoidal trajectory and C++ node. It receives the robot odometry from topic *odom* and publishes the velocity commands on topics *cmd_vel* and *cmd_vel_stamped*;
- record: saves messages from *odom* and *cmd_vel_stamped* topics as bag files.

Chapter 4

Simulations

In this chapter, we show the simulation setup for this work. The first part is related to the trajectories choice and computation of nominal inputs. Then, we present how the controllers are tuned to ensure the best performances for each trajectory. Later, the robot model used in Gazebo simulations is introduced and then the structure of the control node is presented. The chapter ends with a discussion on the simulation results.

4.1 Reference trajectories

Two main approaches can be applied to control the movement of a mobile robot [17]: a point-to-point control and a reference path approach. In the first method only the initial and final pose of the robot are set, while the intermediate path is not important so, in general, this case is used in obstacle-free environments. On the other hand, defining a reference path is more suitable in environments with obstacles, so it is a more appropriate solution in the agriculture framework and it is also the choice made for this works.

In the path control method there is also another distinction between path following and trajectory tracking. The path is defined as a geometric entity representing the series of points that the robot should reach during its movement and can be defined with a function or a set of interpolated way-points. On the other hand, the trajectory is represented by a path plus a time reference that defines at which time each point should be reached. Therefore, for trajectory tracking the robot pose is controlled both in space and time.

In this work we have selected two paths, i.e.

1. Straight line trajectory

2. Sinusoidal trajectory

with a relative velocity profile. These trajectories are representative of a wide range of applications and allow to test the controllers in a scenario similar to a real application.

Once defined the nominal trajectory in term of coordinates $x_r(t)$ and $y_r(t)$, it is possible to derive the reference orientation for the robot $\theta_r(t)$ as described in [44]. Since its heading direction should be tangent to the trajectory in each point, the desired angular position is defined as

$$\theta_r(t) = tg^{-1}(\dot{y}_r(t)/\dot{x}_r(t)) + b\pi \tag{4.1}$$

where b define the motion direction, with $b = 0$ to move forward and $b = 1$ to move backwards. In this study, only the case with $b = 0$ is considered. Then, the reference pose of the robot is defined as $\mathbf{q}_r(t) = [x_r(t), y_r(t), \theta_r(t)]^T$

The common approach for non-holonomic robots is to use a feed-forward action in which robot inputs are obtained from the reference trajectory. The nominal inputs of the robot, i.e. the linear and angular velocities, that are used in the feed-forward action, can be derived as described in [45], [11]. In particular, the reference linear velocity $v_r(t)$ is defined considering the linear velocity vector $\mathbf{v}_r(t)$, obtained from trajectory coordinates i.e.

$$\mathbf{v}_r(t) = (\frac{d}{dt}x(t), \frac{d}{dt}y(t)) \tag{4.2}$$

and then computing the magnitude, i.e.

$$v_r(t) = (-1)^b\sqrt{\dot{x}_r^2(t) + \dot{y}_r^2(t)} \tag{4.3}$$

On the other hand, the reference angular velocity $\omega_r(t)$ is obtained considering for each time instant the movement as a rotation along a circular trajectory, which center is called instantaneous center of curvature and can change at each time instant, as shown in Fig.4.1. In this way the reference angular velocity, is computed as the ratio between the linear velocity, tangential to the path, and the radius of instantaneous curvature $\rho(t)$, i.e.

$$\omega_r(t) = \dot{\theta}_r(t) = \frac{v_r(t)}{\rho(t)} \tag{4.4}$$

The radius $\rho(t)$ is the distance between the instantaneous center of curvature (IcC) and the CoG of the robot, and it is defined as

$$\rho(t) = \frac{(\dot{x}_r^2(t) + \dot{y}_r^2(t))^{3/2}}{\dot{x}_r\ddot{y}_r - \dot{y}_r\ddot{x}_r} \tag{4.5}$$

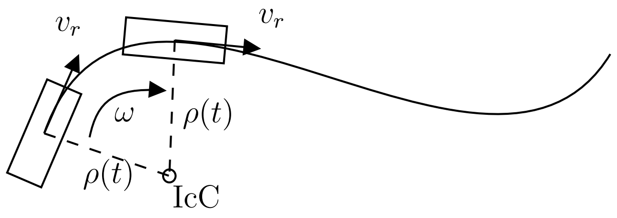


Figure 4.1: Tangential and angular velocity

So, substituting (4.3) and (4.5) in (4.4) the reference angular velocity is given by

$$\omega_r(t) = \dot{\theta}_r(t) = \frac{\dot{x}_r\ddot{y}_r - \dot{y}_r\ddot{x}_r}{\dot{x}_r^2(t) + \dot{y}_r^2(t)} \tag{4.6}$$

From these results, it follows that the desired path should be twice differentiable with respect to time and should have non-zero tangential velocity $v_r(t) \neq 0$. In fact, if for some time instants $v_r(t) = 0$, the robot has only a rotational velocity, and in this case it is not possible to compute the reference orientation.

4.1.1 Straight line trajectory

For straight trajectory the path is defined between the origin ($0m$,$0m$) and point ($2m$,$2m$) and the total path length is $2\sqrt{2}$ m. Considering the maximum linear velocity of the robot, that is $0.2\,\mathrm{ms}^{-1}$, the trajectory is defined such that the velocity is always lower than this limit, i.e.

$$x_r(t) = \begin{cases} 0, & \text{if } t < 1\,\mathrm{s} \\ 0.111(t-1), & \text{if } t \geq 1\,\mathrm{s} \end{cases} \tag{4.7}$$

$$y_r(t) = \begin{cases} 0, & \text{if } t < 1\,\mathrm{s} \\ 0.111(t-1), & \text{if } t \geq 1\,\mathrm{s} \end{cases} \tag{4.8}$$

with initial conditions:

$$[x(0), y(0), \theta(0)] = [0\,\mathrm{m}, 0\,\mathrm{m}, \pi/4\, rad] \tag{4.9}$$

that correspond to the robot orientation aligned with the trajectory in the starting point.

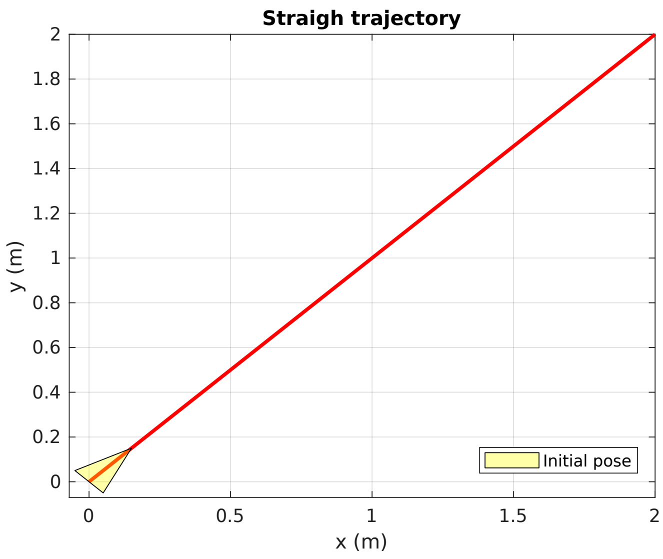


Figure 4.2: Straight reference trajectory, with aligned initial orientation

The choice of starting the trajectory tracking from $t \geq 1\,\mathrm{s}$ is due to ensure that all the controllers are able to start at same time. In fact, after starting the simulation, each of the analysed control approaches takes a different time to reset the Gazebo environment and for starting the data saving. This time is lower than a second, so at $t = 1\,\mathrm{s}$we are able to synchronize all the controllers.

According to the geometry of the path and the velocity profile selected, for the first scenario we have

$$\theta_r(t) = \pi/4\,\mathrm{rad} \tag{4.10}$$

$$v_r(t) = 0.157\,\mathrm{m\,s^{-1}} \tag{4.11}$$

$$\omega_r(t) = 0\,\mathrm{rad\,s^{-1}} \tag{4.12}$$

For the second subcase, the vehicle is initially oriented such that $\theta(0) = 0\,\mathrm{rad}$, as shown in Fig. 4.3. We want to remark that the reference θ_r, v_r and ω_r for this case are the same of the previous one.

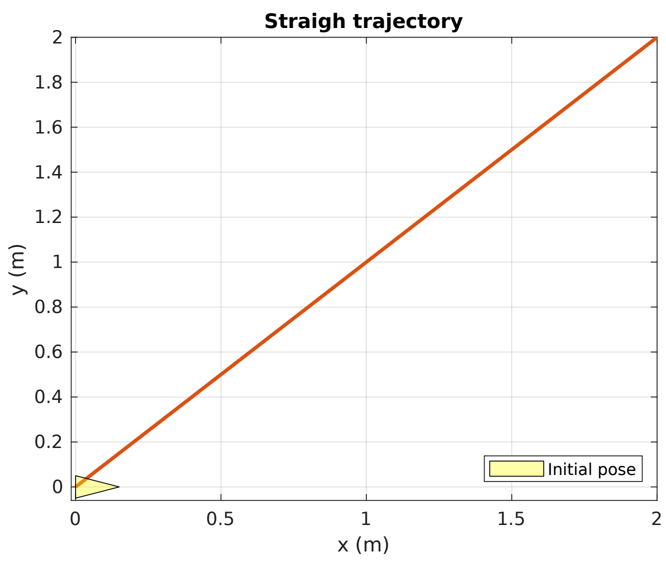


Figure 4.3: Straight reference trajectory, with non-aligned initial orientation

4.1.2 Sinusoidal trajectory

For the second scenario, a sinusoidal trajectory is defined, i.e.

$$x_r(t) = \begin{cases} 0, & \text{if } t < 1\,\text{s} \\ 0.07(t-1), & \text{if } t \geq 1\,\text{s} \end{cases} \tag{4.13}$$

$$y_r(t) = \begin{cases} 0, & \text{if } t < 1\,\text{s} \\ 2sin(x(t)), & \text{if } t \geq 1\,\text{s} \end{cases} \tag{4.14}$$

with initial conditions:

$$[x(0), y(0), \theta(0)] = [0\,\text{m}, 0\,\text{m}, 1.1071\,rad] \tag{4.15}$$

that corresponds to the robot orientation aligned with the trajectory in the starting point.

Also in this case, the trajectory is defined for $t \geq 1\,\text{s}$, and is chosen to ensure the limitation of robot's maximum speed.

From (4.1) it is possible to compute the reference orientation as

$$\theta_r(t) = tan^{-1}(2cos(0.07(t-1))) \tag{4.16}$$

Whereas from (4.3) and (4.6) reference linear and angular velocities can be obtained as

$$v_r(t) = \sqrt{0.07^2 + (0.14cos(0.07(t-1)))^2} \tag{4.17}$$

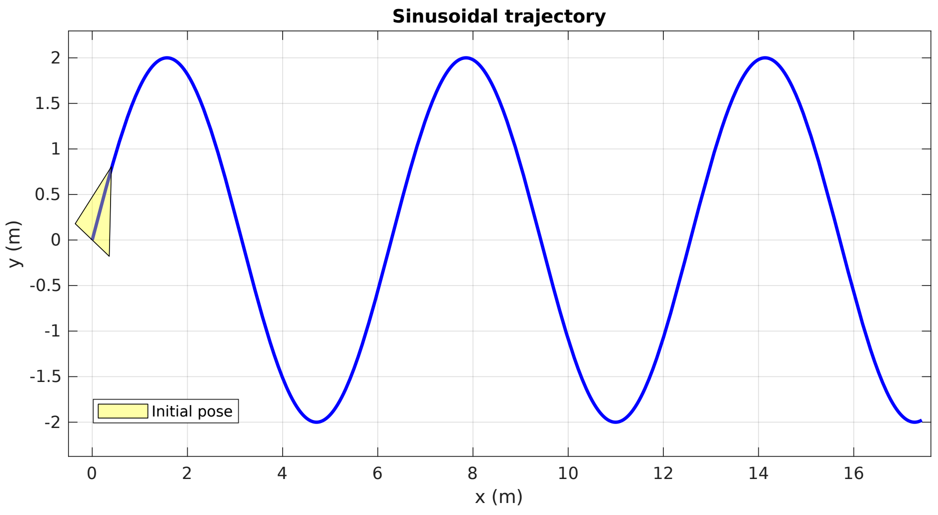


Figure 4.4: Sinusoidal reference trajectory

$$\omega_r(t) = \frac{-6.86 \cdot 10^{-4} sin(0.07(t-1))}{0.07^2 + 0.14cos(0.07(t-1))^2} \tag{4.18}$$

The obtained references are shown in Fig. 4.5

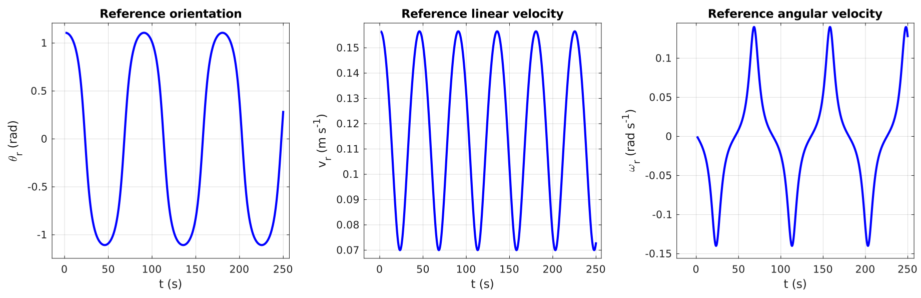


Figure 4.5: In order: Reference direction, Reference linear velocity, Reference angular velocity for sinusoidal trajectory

We can notice that the maximum value of linear and angular velocities are within the robot's limits, i.e. $0.2\,\mathrm{ms}^{-1}$ for linear velocity and $0.5\,\mathrm{rad\,s}^{-1}$ for the angular one.

4.1.3 PID and LQR parameters tuning

Controllers performances depend on the value of their tuning parameters, but the behaviour changes also according to the chosen reference trajectory. For this reason a proper tuning should be performed for each scenario.

Each controller has its own tuning techniques, as shown in [46] for PID or [47] for LQR , but the simplest one is the trial-and-error method, that has been exploited in this work.

4.2 PID tuning

The tuning coefficients of PID controller are the parameters: K_p, K_i, K_d. This tuning process is based on an initial tuning of the proportional coefficient K_p, with the other ones equal to zero, considering only the proportional action at a first step. At this point it is find a value of K_p with a reasonable magnitude, chosen also with respect to the coefficients of the other PIDs of the system. After that, the integral coefficient K_i is tuned, choosing a value that ensures a good reduction of tracking error. In the end, the values of differential coefficient K_p is tuned, being able to reduce the oscillatory behaviour.

The resulting values are reported in Tab 4.1

Coefficient:	Sinusoidal	Straight 45°	Straight 0°
K_{p_x}	0.2	0.7	6
K_{i_x}	0.02	0.03	0.1
K_{d_x}	0.02	0.01	0.1
K_{p_y}	0.05	0.05	6
K_{i_y}	0.2	0.04	0.3
K_{d_y}	0.01	0.0	0.1
K_{p_θ}	1	6	1
K_{i_θ}	0.2	0.3	0.1
K_{d_θ}	0.2	0.05	0.05

Table 4.1: Table of PID coefficients chosen for each trajectory

LQR tuning

LQR is tuned designing the penalty weight matrices **Q** and **R**, assuming that they are both diagonal matrices, i.e.

$$Q = \begin{bmatrix} q_{11} & 0 & 0 \\ 0 & q_{22} & 0 \\ 0 & 0 & q_{33} \end{bmatrix} \tag{4.19}$$

$$R = \begin{bmatrix} r_{11} & 0 \\ 0 & r_{22} \end{bmatrix} \tag{4.20}$$

As shown in Section 2.3, each diagonal element defines the contribution of a single state or input, in particular q_{11} is related to the contribution of the error on x, q_{22} to the error on y and q_{33} to the error on θ. On the other hand, r_{11} is related to the contribution of command input u_v and r_{22} to command input ω_r.

As starting values for **Q** and **R** are used identity matrices. Since all elements have same magnitude there is no parameters with higher priority. After this first choice, the values are tweaked to achieve better performances. In particular values of **Q** are used to reduce the error of robot's pose and using higher values for some components allows to give priority to that pose components. On the other hand, values of **R** are used to reduce the control effort, since using higher values allows to reduce the control magnitude.

In the tuning, it is not important the magnitude of the matrix entries itself, but the relative magnitude between weights. Using higher values for **Q** allows to have better tracking performances, but the control effort is enlarged at same time. Instead, increasing values for **R** reduces the control input magnitude, but also reduces the system's reactivity, so a good choice is a trade-off between accuracy and control effort.

For tuning the LQR parameters, a Simulink model was used before implementing the controller in the ROS node. The block scheme used in Simulink is reported in Fig. 4.6.

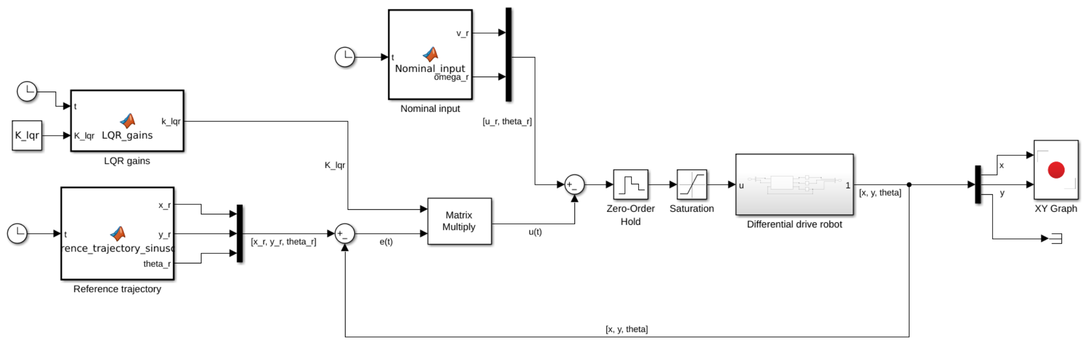


Figure 4.6: Simulink implementation of LQR controller

The values chosen in the tuning process are shown in Tab 4.2 We can notice that for sinusoidal and straight trajectories, to reach an acceptable tracking precision, it is sufficient to use the same magnitude for each components of **Q**, while the elements of **R** are chosen two orders of magnitude lower to ensure a good reactivity. Instead, for the straight trajectory with initial orientation $\theta(0) = 0\ deg$, the two components of **Q** relative to x and y are chosen 20 times higher than the one of θ.

Coefficient:	Sinusoidal	Straight 45°	Straight 0°
q_{11}	1	1	1
q_{22}	1	1	1
q_{33}	1	1	0.05
r_{11}	0.01	0.01	0.01
r_{22}	0.01	0.01	0.01

Table 4.2: Table of LQR weights chosen for each trajectory

This is done to ensure the reduction of inaccuracy on x and y, compensating the initial error due to non-aligned original orientation.

4.3 Simulation results

In this section, the simulation results obtained from the two scenarios, implementing the selected control strategies in the different languages, are presented. To ensure the repeatability of the results, 10 simulations have been run for each setup.

The different case studies are then compared in terms of

- actual $(x(t), y(t))$, vs desired $(x_r(t), y_r(t))$ trajectory
- the tracking error, defined as

$$e_t(t) = \sqrt{(x_r(t) - x(t))^2 + (y_r(t) - y(t))^2} \tag{4.21}$$

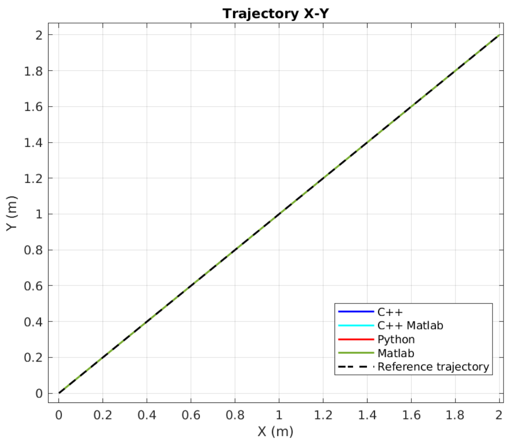
- the command inputs for linear and angular velocity, computed by the control algorithm
- the execution times needed to compute each control input

On the same graph are superimposed the plot of each programming method, to compare the behaviours.

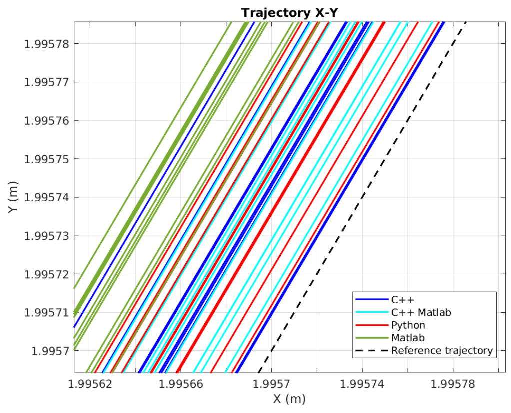
4.3.1 Straight line trajectory, with $\theta = \pi/4$ rad and controlled with a PID

The first case study is the straight line trajectory, considering the case of initial orientation aligned to the reference path and, as tracking controller, the PID is applied .

The trajectory plot is shown in Fig. 4.7 and from this it is possible to notice that all the controllers are able to track the reference trajectory from initial point to final one. From the figure's zoom around the final point, it can be noticed that C++, Python and C++ Matlab trajectories are more close to the reference one in most of cases, while Matlab simulations are further, resulting the one to have the worst tracking performances.



(a) Trajectory x-y



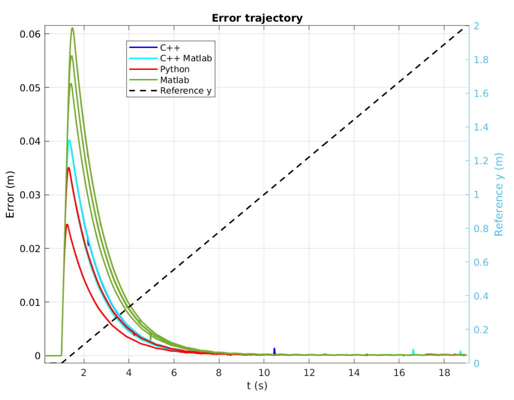
(b) Trajectory x-y zoom

Figure 4.7: Straight, $\pi/4$ *rad*, PID: trajectory

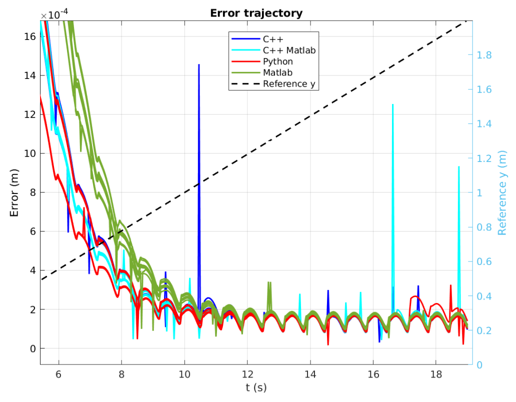
These results are confirmed by the tracking error plot in Fig. 4.8, where it is possible to notice that there is a larger error at the beginning of simulation due to the effect of the controller (overshoot). After approximately 8 seconds, the controller is able to steer the system to the desired trajectory, with a resulting drop of the tracking error to the order of magnitude of $10^{-4}m$ for all the controllers. In this part, the tracking error has a small ripple that is common to all the controllers. There are also some error peaks in other points, that are not repetitive, and depends on the single simulations. Comparing the tracking error of the different controllers, it is possible to notice some differences in the first part of trajectory with Python that presents a slightly smaller error, while Matlab has the larger one. Instead, when the robot converges to the reference trajectory all controllers have the same error.

For the command inputs in Fig. 4.9, we can notice an initial effort, higher in both linear and angular commands. This correspond to the initial error shown in the previous graph, because the controller tries to react increasing the control action.

In this graph, and also in the others related to the command inputs, there are



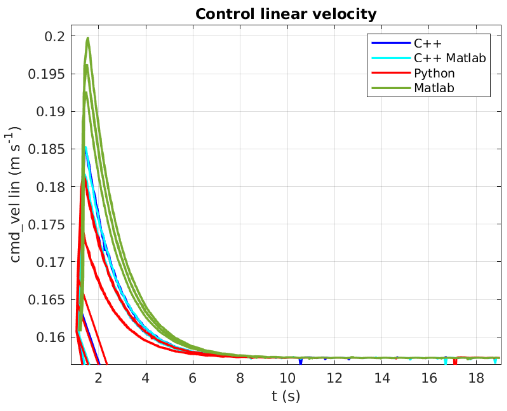
(a) Trajectory error



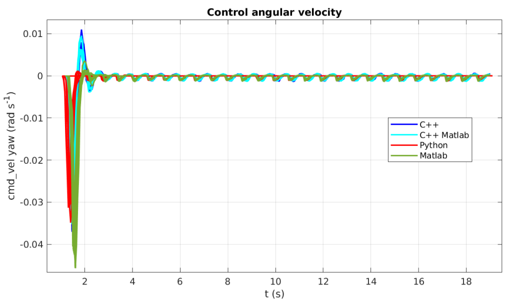
(b) Trajectory error zoom

Figure 4.8: Straight, $\pi/4$ *rad*, PID: Tracking error

some miss-placed points. The cause is an error in the function used to get the simulation time in command messages timestamps, i.e. in some occasions the time value in the time stamp is wrong, and this cause the presence of random positioned points.



(a) Command linear velocity



(b) Command angular velocity

Figure 4.9: Straight, $\pi/4$ *rad*, PID: Linear and angular velocity commands

Comparing the execution time of the controllers in Fig. 4.10, it is possible to notice the major differences among the languages. The controller based on C++ Matlab has a lower execution time with respect to all the others. This difference is shown also in Table 4.4, where the mean values for each controller are reported. In particular, the C++/Matlab based controller is about 10 times faster than the on in C++. Matlab and Python have similar values for execution time. A difference between Matlab-Python and C++ was expected, since the first ones are interpreted programming languages, while the second is a compiled one, but what we can observe is the difference between C++ and C++ Matlab, because both of them are compiled in the same way. The main reason of this difference is the Matlab's code optimization, that allows to reach this performances.

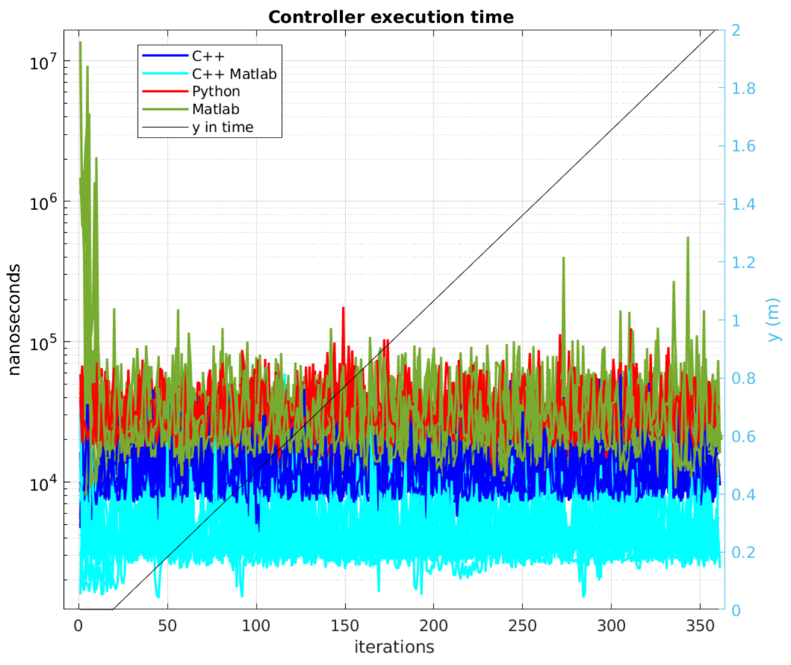


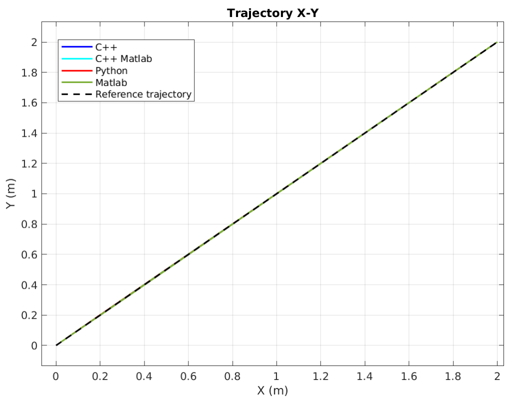
Figure 4.10: Straight, $\pi/4$ *rad*, PID: execution time

4.3.2 Straight line trajectory, with $\theta = \pi/4$ *rad* and controlled with a LQR

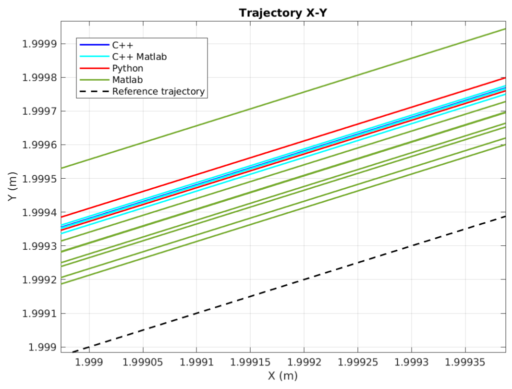
In this section, we report the results obtained considering the LQR controller for the same case study.

The Fig. 4.11 shows a good tracking performance of the controller as for the PID case. On the zoom around the final point, it is possible to notice some differences with respect to the previous case, i.e. the trajectories of C++, Python and C++ Matlab are more overlapped with respect to the PID case. It means that their performances are more repeatible over multiple simulations. For what concerns the Matlab case, it has trajectories completely different from each other, and this allows to be also closer to the reference in some cases, but it has a less repeatable behaviour.

The trajectory error in Fig. 4.12 has the same global behaviour of previous PID, with a larger error in the first part that is then reduced. The difference is that in



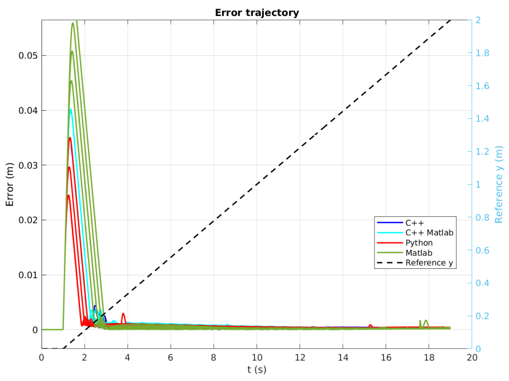
(a) Trajectory x-y



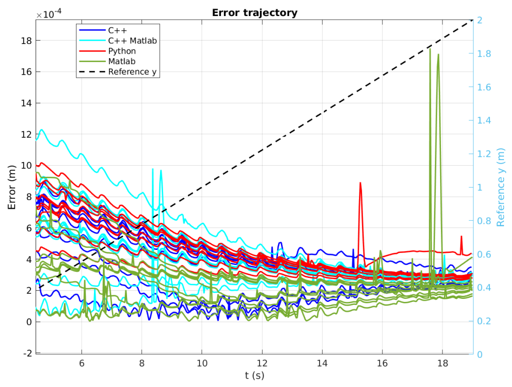
(b) Trajectory x-y zoom

Figure 4.11: Straight, $\pi/4$ rad, LQR: trajectory

this case the converging time is lower, beeing about 3 second versus 8 second of previous case. Considering the error behaviour after the convergence, the LQR has an higher variation in the error, that is still in the order of magnitude of 10^{-4}m as the PID.



(a) Trajectory error

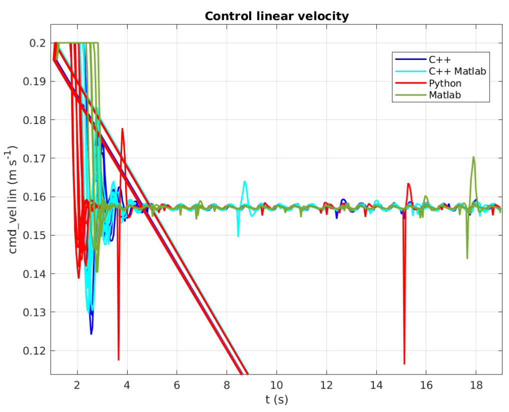


(b) Trajectory error zoom

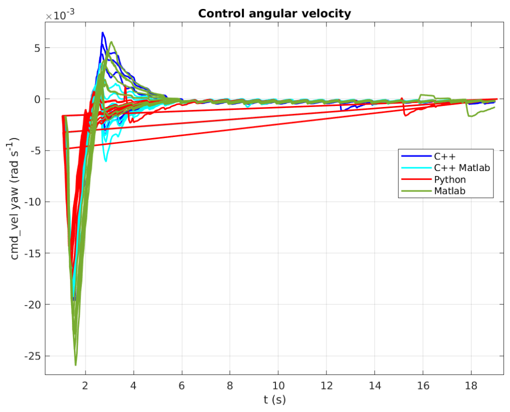
Figure 4.12: Straight, $\pi/4$ rad, LQR: tracking error

The command input reported in Fig. 4.13 presents an higher value of linear and angular velocity in the first part, with respect to the previous case. This is due to an higher tracking error in the first part of the trajectory with respect to the PID case.

Execution time in Fig. 4.14 confirms the good performances of C++ Matlab,



(a) Command linear velocity



(b) Command angular velocity

Figure 4.13: Straight, $\pi/4\ rad$, LQR: Linear and angular velocity commands

that has the lower value also in this case, but there is an important difference in the Matlab and C++ behavior, i.e. Matlab has better performances than C++. The reason could be that Matlab code optimization, in the case of LQR algorithm, has a larger impact with respect to the benefits of a compiled-language as C++. A confirm to this hypothesis is that Python has worse performances while C++ Matlab, that has the optimized code and it is also compiled, improves the performances of Matlab having better results.

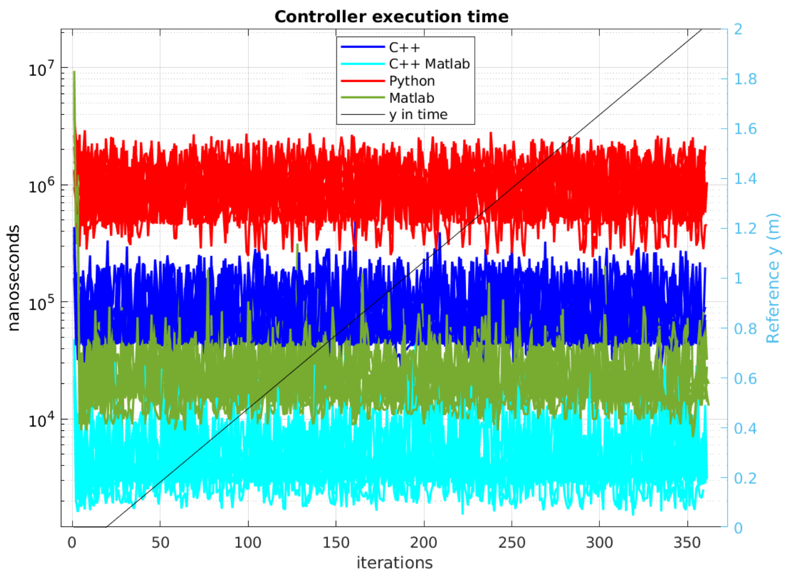
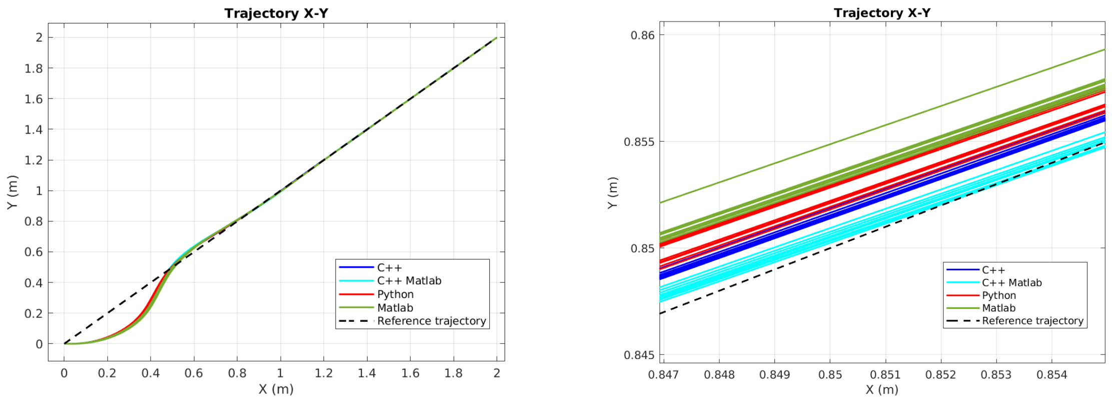


Figure 4.14: Straight, $\pi/4\ rad$, LQR: execution time

4.3.3 Straight line trajectory, with $\theta = 0\ rad$ and controlled with a PID

The next case study considers the straight trajectory with initial condition not aligned to the reference trajectory, i.e. initial orientation of 0 deg.

The controlled trajectories are reported in Fig. 4.15. The first part of the trajectory is characterized by the curvature that leads the robot to the reference path, before converging to it. When the robot is approaching the nominal trajectory, it is possible to notice an overshoot, that leads to an error in the opposite direction, before the definitive convergence. The zoom section is focused around the convergence point and it shows that the C++ Matlab case get closer to the reference, with a faster convergence with respect to the others.

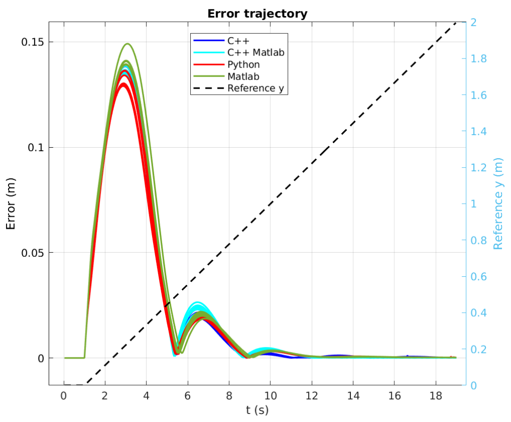


(a) Trajectory x-y **(b)** Trajectory x-y zoom

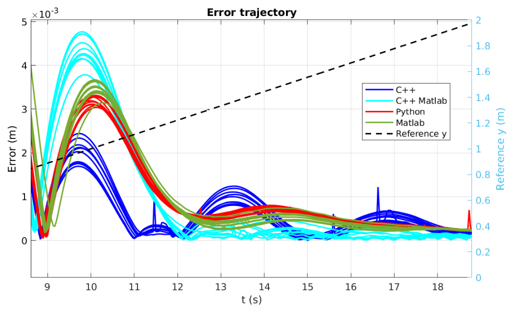
Figure 4.15: Straight, 0 rad, PID: trajectory

The error plot in Fig. 4.16 reflects the trajectory behaviour. There is an initial low error, since the robots initial position is along the reference trajectory. Then, the error is increased since the robot is moving away due to the initial orientation. After reaching the maximum, the error reduces again when the robot is approaching the reference. At this point, there is another error peak, that corresponds to the overshoot of the robot. The error values after convergence are slightly larger than the trajectory with aligned initial condition, but in both cases veery small values of the tracking error are achieved.

For what concerns the control commands, shown in Fig.4.17, it is possible to notice that there is an initial saturation of both linear and angular commands, due to the initial effort that is needed to align the robot. When the robot is approaching the reference trajectory, the command is reduced, while later increases again in



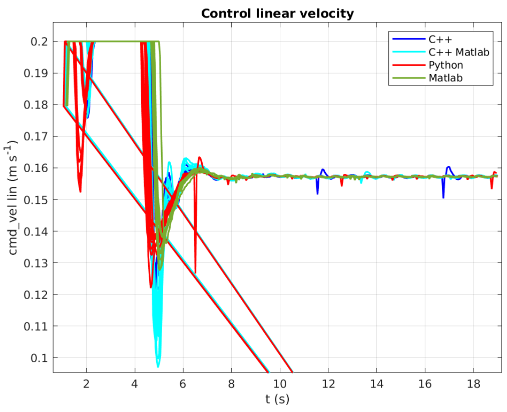
(a) Trajectory error



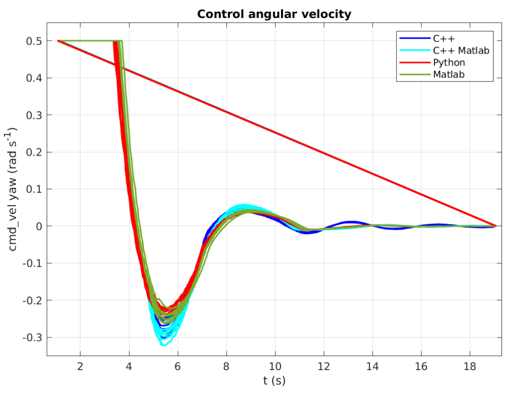
(b) Trajectory error zoom

Figure 4.16: Straight, 0 rad, PID: tracking error

correspondence of the overshoot. With respect to the aligned case, now we have a larger command action, as expected.



(a) Command linear velocity



(b) Command angular velocity

Figure 4.17: Straight, 0 rad°, PID: Linear and angular velocity commands

The execution time in Fig. 4.18 shows similar results to the aligned case with PID, with better performances of C++ Matlab with respect to C++, Python and Matlab. This case confirms that with PID controllers Python and Matlab time performances are the worst, while with the previous LQR Matlab is better than C++.

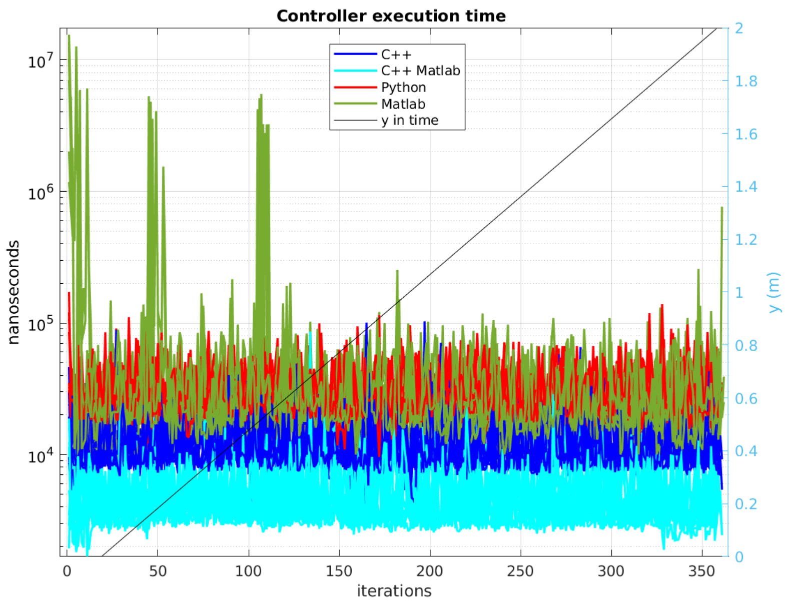
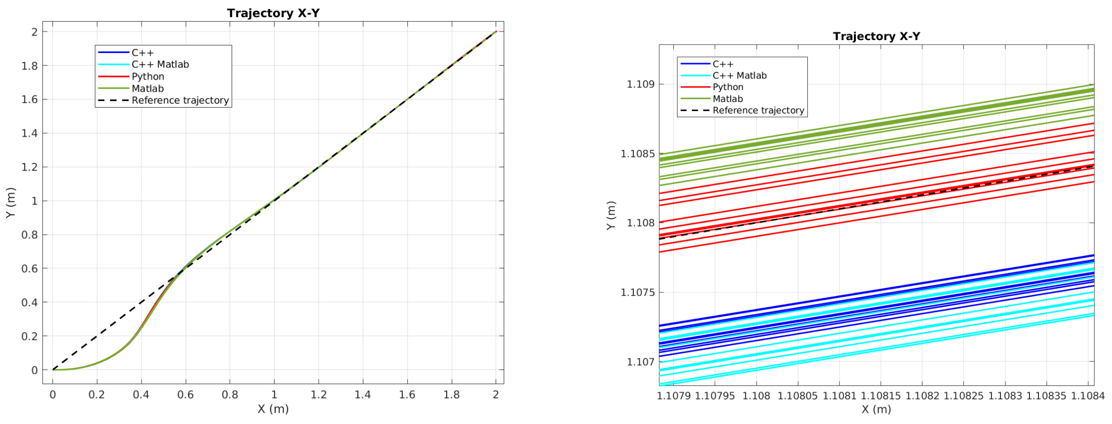


Figure 4.18: Straight, 0 rad, PID: execution time

4.3.4 Straight line trajectory, with $\theta = 0\ rad$ and controlled with a LQR

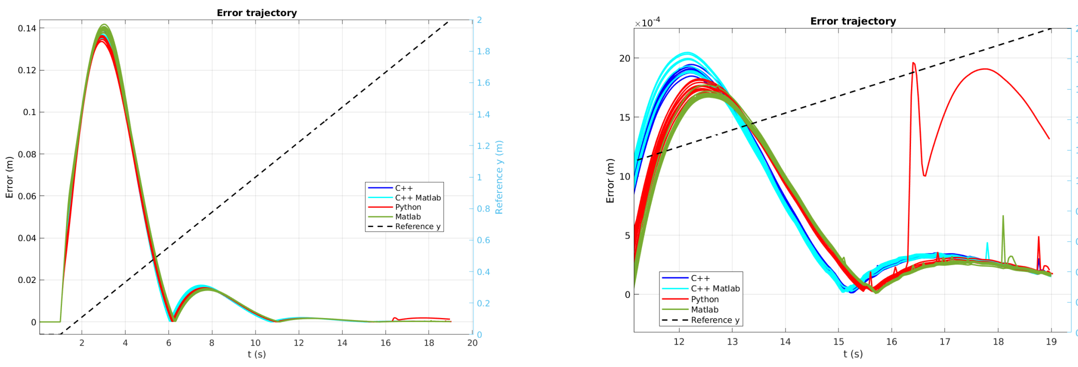
The PID case is then comapared with the LQR, considering the same trajectory setup. Trajectory in Fig. 4.19 has same behaviour of PID case, with the same overshoot when approaching the reference. Also LQR tracking error in Fig. 4.20 is



(a) Trajectory x-y **(b)** Trajectory x-y zoom

Figure 4.19: Straight, 0 rad, LQR:trajectory

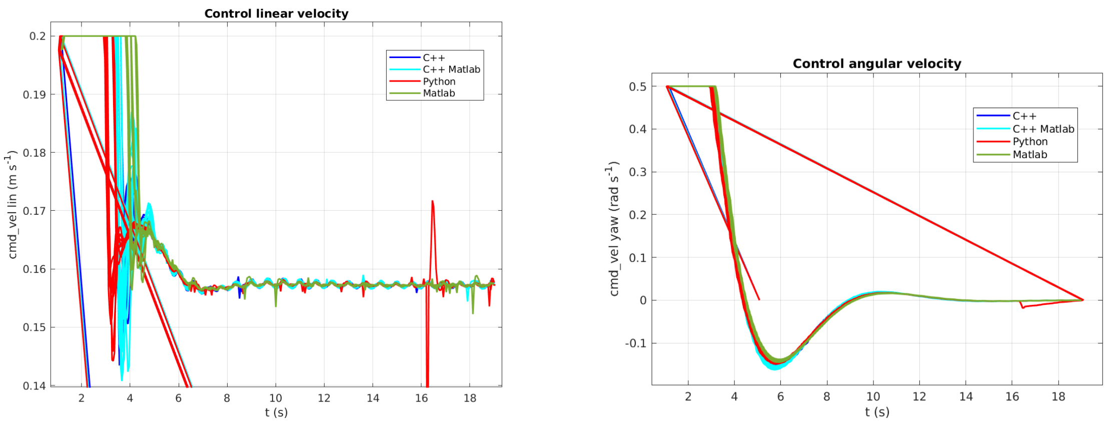
similar with respect to PID case, with the same order of magnitude of the error after convergence, and a similar convergence time. Control commands in Fig. 4.21



(a) Trajectory error **(b)** Trajectory error zoom

Figure 4.20: Straight, 0 rad, LQR: tracking error

presents an initial saturation, as in PID case. For the execution time in Fig. 4.22,



(a) Command linear velocity **(b)** Command angular velocity

Figure 4.21: Straight, 0 rad, LQR: Linear and angular velocity command

there is the same trend already observed in previous case with LQR, with Matlab execution time that is comparable with C++, and it is better than Python, thanks to the code optimization.

4.3.5 Sinusoidal trajectory, controlled with a PID

The next trajectory is the sinusoidal one, controlled using a PID controller.

The trajectory plot is depicted in Fig. 4.23. From a global point of view, the controller is able to track the reference path for the entire simulation. From the

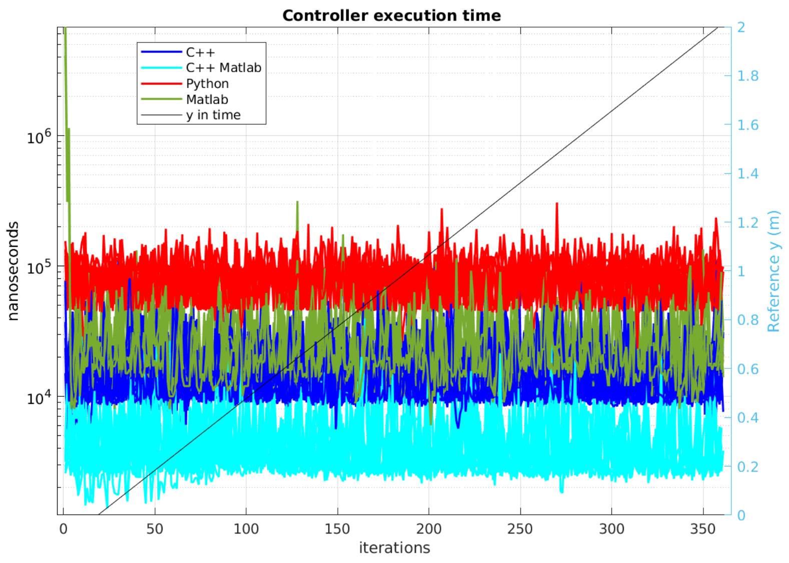
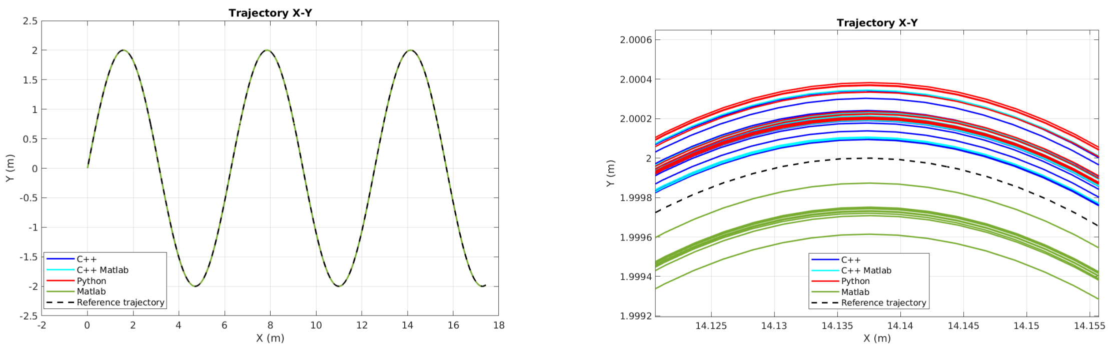


Figure 4.22: Straight, 0 rad, LQR: execution time

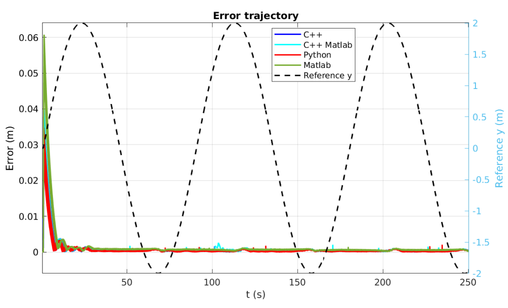
zoom around the last peak of the sinusoidal path, it is possible to notice that Matlab simulations have the lower tracking error, while all the other controllers have higher values. This difference between Matlab and the others controllers is



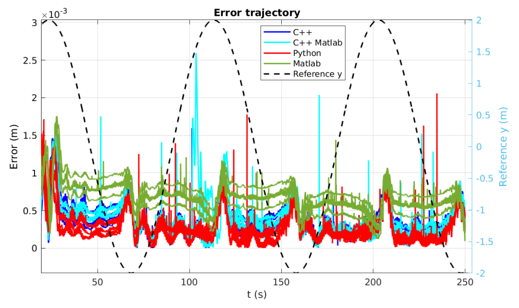
(a) Trajectory x-y **(b)** Trajectory x-y zoom

Figure 4.23: Sinusoidal, PID: trajectory

clear also from the error plot in Fig. 4.24. Considering the zoom section, the Matlab graphs have higher values with respect to the others, and these graphs are also quite different between each other, highlighting a less repeatable behaviour. The other controllers are more aligned and have similar graphs, and this is also confirmed from the data reported in Tab. 4.3. In the same table, we can notice that Python has a slightly better performance, with lower variance, but the difference is very small with respect to the C++ and C++ Matlab cases.



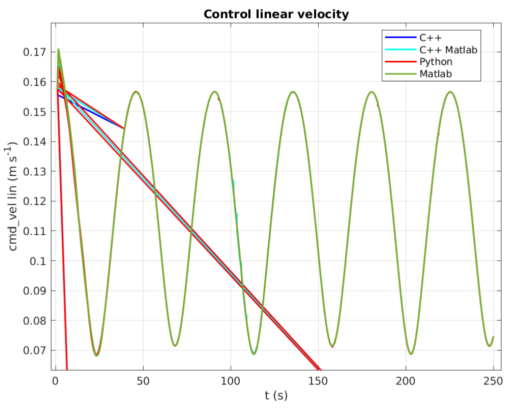
(a) Trajectory error



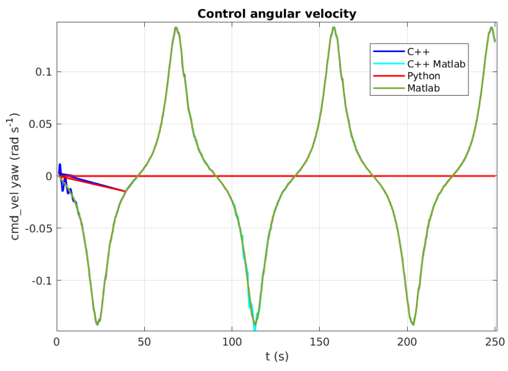
(b) Trajectory error zoom

Figure 4.24: Sinusoidal, PID: tracking error

The plot of the system inputs are shown in Fig.4.25. It is possible to notice an initial larger action in the first part of the trajectory due to the initial effort to correct the error, like for the previous controllers. Then, the input becomes similar to the theoretical one represented in Fig. 4.5.



(a) Command linear velocity



(b) Command angular velocity

Figure 4.25: Sinusoidal, PID: Linear and angular velocity command

In the execution time plot in Fig. 4.26, the behaviour of the straight line cases is confirmed, with smaller values for the C++ Matlab case. The other controllers have similar values between each other, with quite overlapping graphs and this is a common aspect to all PID applications.

4.3.6 Sinusoidal trajectory, controlled with a LQR

The last case is the sinusoidal trajectory where the robot is controlled using LQR.

The trajectories are reported in Fig. 4.27. In the zoom section, we can observe

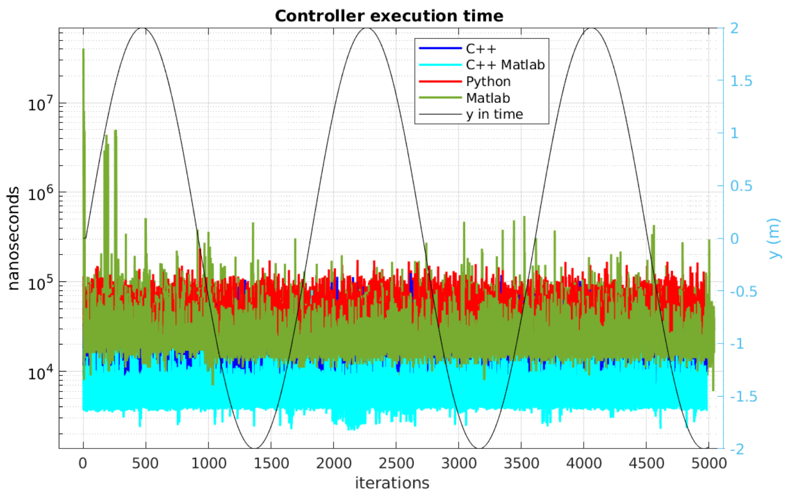
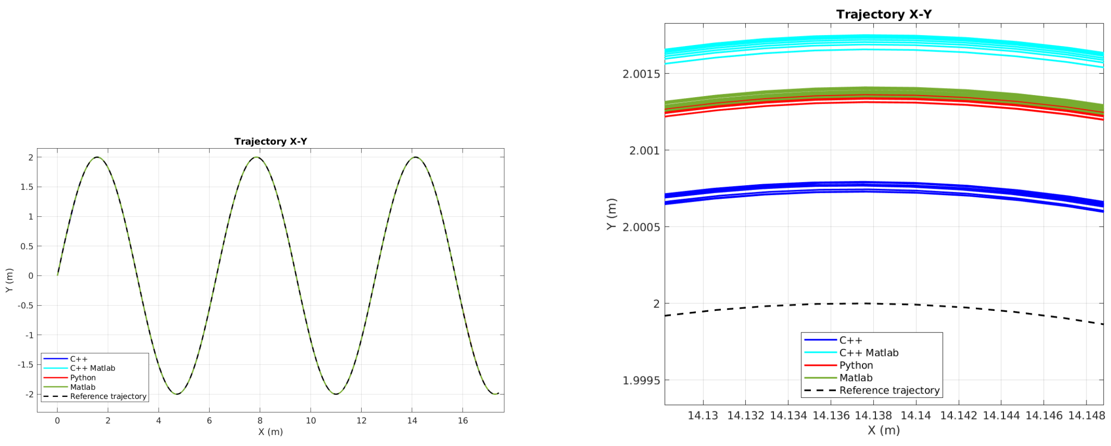


Figure 4.26: Sinusoidal, PID:: execution time

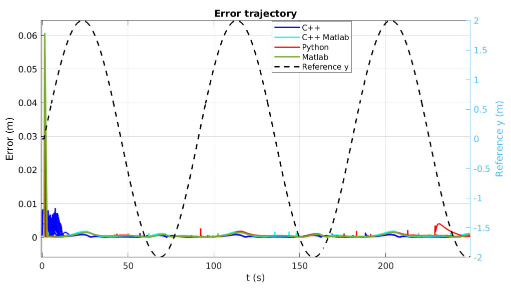
a difference with respect to the PID case, i.e. the trajectories of each controller are quite overlapped each others, showing high repeatability. This is a common feature of all the LQR cases, excluding some exceptions in the Matlab case. In addiction, Matlab and Python are quite similar.



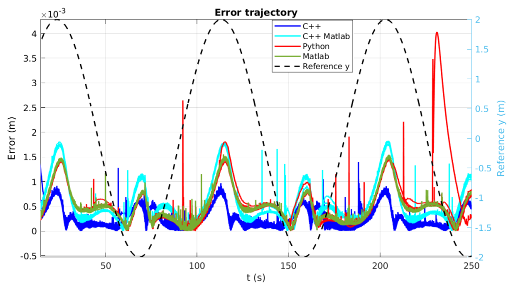
(a) Trajectory x-y **(b)** Trajectory x-y zoom

Figure 4.27: Sinusoidal, LQR: trajectory

The tracking error in Fig. 4.28 shows a better performance of C++ case, while other controllers have slightly higher values. Another case in which C++ showed the lower error is in the straight 0 rad trajectory with PID controller, so this behaviour is independent by the controller type. In the first part of the trajectory, C++ showed also a more oscillating behavior with respect to the others.



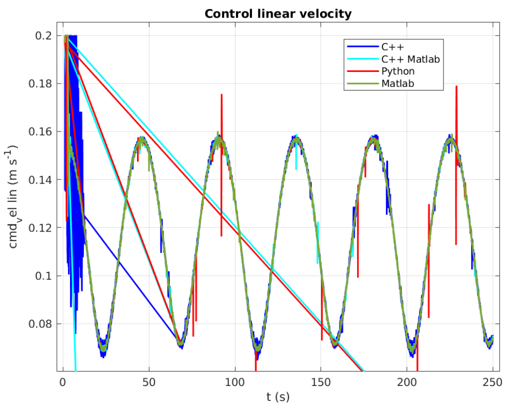
(a) Trajectory error



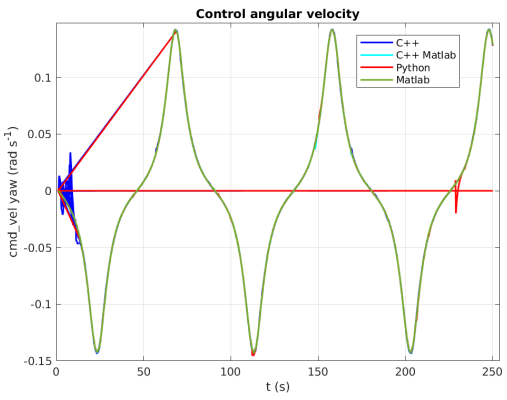
(b) Trajectory error zoom

Figure 4.28: Sinusoidal, LQR: tracking error

The oscillating behaviour of C++ is more evident in commands plot in Fig. 4.29, where in both linear and angular commands there are oscillations, mainly in the first part of trajectory. The oscillations of C++ are present also in the PID case of sinusoidal tracking, but are less evident. Another difference is the presence of some peaks for the Python case, mainly in the linear command, related to the error peaks shown in the error plot, but they are not repetitive and depend by the execution. This behaviour can be find also in the straight $\pi/4$ rad using the LQR.



(a) Command linear velocity



(b) Command angular velocity

Figure 4.29: Sinusoidal, LQR: Linear and angular velocity command

The execution time graph in Fig. 4.30 shows a similar behavior with other LQR cases, with better performances of Matlab with respect to C++, while C++ Matlab remains the faster approach.

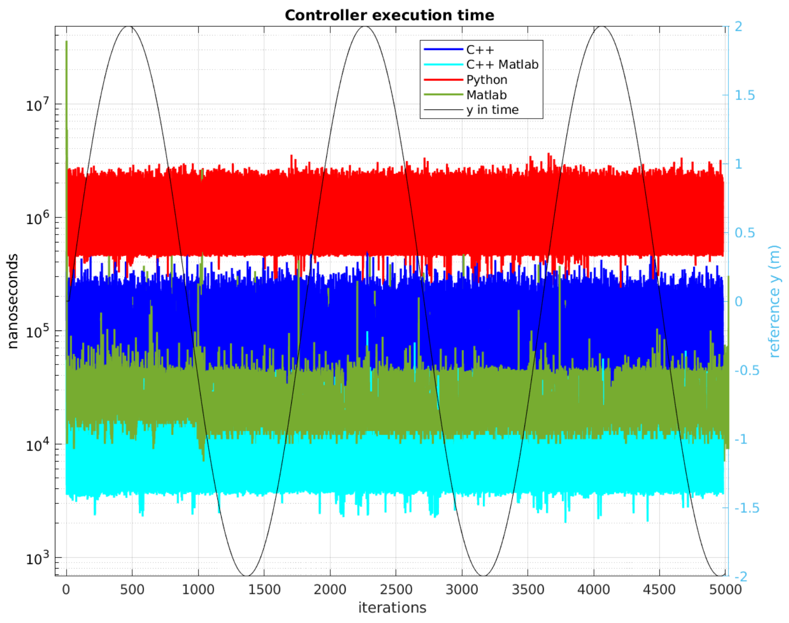


Figure 4.30: Sinusoidal, LQR: execution time

4.3.7 Final analysis

Tracking error performances of simulations can be summarized considering the mean value and variance of the error along the 10 simulations. The values are reported in Tab. 4.3. From mean value comparison, it can be noticed that straight

		PID				LQR			
Traj		C++	Pyt	C++ Mat	Mat	C++	Pyt	C++ Mat	Mat
Str $\pi/4$	$\mu[10^{-4}m]$	32	25	30	51	17	13	17	27
	$\sigma[10^{-6}m^2]$	55.10	31.82	47.33	131.5	31.85	17.82	30.45	84.21
Str 0	$\mu[10^{-4}m]$	213	208	217	233	234	235	235	251
	$\sigma[10^{-6}m^2]$	1500	1400	1500	1700	1600	1600	1600	1700
Sin	$\mu[10^{-4}m]$	8.52	6.94	8.40	14.00	4.36	5.61	6.56	6.98
	$\sigma[10^{-6}m^2]$	9.05	6.80	8.97	23.40	3.02	1.39	2.71	8.33

Table 4.3: Table of mean and variance of tracking error

line trajectory with aligned initial position has an higher error with respect to the sinusoidal one. The main reason is that the first one is travelled with an higher speed with respect to the second, so this penalizes the tracking performances. On the other hand, a higher error for the straight trajectory non-aligned is expected due to its initial conditions. If we compare the mean value of PID and LQR controllers, we can notice that, with PID, Python has slightly better performances, while Matlab is the worst. C++ and C++ Matlab have similar values, that are close to the Python ones. For LQR, Matlab has worst performances, while the difference between the other controller is less evident.

Finally, the execution time performances are summarized in Table 4.4, where the mean value, the minimum and maximum execution times are reported. We can see

		PID				LQR			
Traj		C++	Pyt	C++ Mat	Mat	C++	Pyt	C++ Mat	Mat
Str $\pi/4$	$\mu[10^4 ns]$	1.21	2.83	0.46	6.71	7.83	107.66	0.48	4.39
	min $[10^4 ns]$	0.44	0.98	0.15	0.8	2.74	24.11	0.15	0.7
	max $[10^4 ns]$	6.05	17.7	6.27	1378.1	52.85	368.4	4.82	940
Str 0	$\mu[10^4 ns]$	1.20	2.86	0.47	10.43	1.48	8.37	0.43	4.67
	min $[10^4 ns]$	0.41	0.96	0.16	0.7	0.53	2.33	0.14	0.6
	max $[10^4 ns]$	10.29	17.16	8.59	1551.7	11.8	30.65	3.97	973.6
Sin	$\mu[10^4 ns]$	1.47	3.70	0.66	3.30	8.34	112.99	0.71	2.82
	min $[10^4 ns]$	0.52	1.18	0.22	0.60	2.80	23.87	0.20	0.60
	max $[10^4 ns]$	12.60	23.52	8.94	4047.7	60.47	369.21	12.40	3596.7

Table 4.4: Table of execution time, with mean, minimum and maximum values

that C++ Matlab has the lowest times. This is valid for both PID and LQR, while for the other controllers there are some differences between the two approaches, i.e. in PID, C++ has the second best performances, while in LQR Matlab has better performances than C++ and Python. Instead, in the comparison between C++ and Python, C++ is better in all cases.

From these results, it is clear that, to obtain better performances from the execution time point of view, the best approach is C++ Matlab, since for all the controllers it has the best results. The benefits of Matlab's code optimization are also evident, not only in the final C++ code, but also running the controller directly on Matlab, especially in the case of a more complex controller. On the other hand, from a tracking error point of view, Matlab has slightly worse results, mainly due to a low repeatability between simulations, but the magnitude of this difference is small compared to the whole trajectory.

In conclusion, the integration of Matlab in ROS introduces many advantages for the controller design and its optimization and is an opportunity to be exploited.

Chapter 5

Conclusion and future work

The goal of this thesis research was the comparison of different approaches to the development of ROS control systems. This has been reached comparing the most popular programming languages for ROS, i.e. C++, Python, Matlab and C++ obtained from Matlab coder, used to implement a PID and LQR controller.

The main focus was on the comparison between C++/Python, that represents the classical approach with ROS, and Matlab/C++ Matlab, that allows to extend the ROS functionalities with the integration of the Matlab environment.

This comparison was made realizing different simulations in the Gazebo environment and using as robot model a DDMR, so first of all its mathematical model has been derived. Then two reference trajectories have been chosen, i.e. a straight line trajectory and a sinusoidal one, and the controllers have been designed and tuned accordingly.

Simulation performances are compared from two points of view: tracking performances and control execution time, as shown from graphs in Section 4.3.

The results obtained demonstrate that, for what concern the tracking error, all the approaches have reached similar results, with only Matlab that in some cases has worse performances, but the differences are small with respect to the whole trajectory magnitude.

The main results regard the execution time, since, from all the tests, it is clear that C++ Matlab has the best performances, with a large difference with respect to all the other cases. Another interesting result is related to the Matlab time performances, since in the LQR case it has better or similar performances with respect to C++. This underlines the Matlab's code optimization, that represents an advantage of the integration between ROS and Matlab. In this way it shows that it is possible to perform a first test using the code written in Matlab, running the simulation from it, and when the code development is terminated, it is possible to export the code in C++, obtaining also better performances from the execution time point of view. This advantage is more evident with a more complex controller, as can be seen from the difference between PID and LQR, where in the first case the convenience of Matlab is not evident.

Regarding the Python controller, the small better performances in trajectory tracking does not justify the much larger execution times in all cases.

Future developments of the proposed research could aim at test more complex controllers from the computation point of view, to see the effect on execution time and understand if the benefits of Matlab code optimization are still evident.

Another adding to the work can be the test on a real robot, moving to the hardware-in-the-loop approach, to see if the obtained results are still valid and how performances changes in a real application. It is necessary to focus especially on the tracking error, since with simulations we reached good results, with errors in the order of magnitude of $10^{-4}m$, but in a real scenario is difficult to reach this precision, mainly due to sensors accuracy, noise and non-linearities.

Appendix A

ROS Manual files

A.0.1 Launch file example

```
    <launch>

  <!--  Reset Gazebo -->
  <node name="global_loc" pkg="rosservice" type="rosservice"  args="
    call --wait /gazebo/reset_simulation" />


 <!-- ROSBAG: -p: print when start saving, -O: file location,     -
    b: buffer size        duration: of savage,  /:topic to save  -->
   <node name="record" pkg="rosbag" type="record" args="-p  -O /home/
    gerardo/catkin_ws/src/pid_controller/bagfiles/rosbag_cpp_sin.bag
    --duration=250      odom cmd_vel_stamped" />


  <!-- Controller node -->
  <node name="nodo_sin_cpp" pkg="pid_controller" type="cpp_pid_sin"
    output="screen"/>

</launch>
```

A.0.2 Message file example

```
  geometry_msgs/TwistStamped.msg

    std_msgs/Header header
    geometry_msgs/Twist twist
```

```

std_msgs/Header.msg
     uint32 seq
     time stamp
     string frame_id

geometry_msgs/Twist.msg
     geometry_msgs/Vector3 linear
     geometry_msgs/Vector3 angular

geometry_msgs/Vector3.msg
     float64 x
     float64 y
     float64 z

nav_msgs/Odometry.msg
     std_msgs/Header header
     string child_frame_id
     geometry_msgs/PoseWithCovariance pose
     geometry_msgs/TwistWithCovariance twist
```

A.0.3 Matlab launcher icon. Content of file "matlab_r2022a.desktop"

```
[Desktop Entry]
Version=1.0
Type=Application
Name=MATLAB R2022a
Terminal=false
Exec=env MESA_LOADER_DRIVER_OVERRIDE=i965 matlab22 -desktop
Icon=matlab
Categories=Development;Math;Science
Comment=Scientific computing environment
StartupNotify=true
StartupWMClass=com-mathworks-util-PostVMInit
```

Bibliography

[1] Jelle Bruinsma. *World agriculture: towards 2015/2030: an FAO perspective.* Routledge, 2017 (cit. on p. 1).

[2] Christophe Cariou, Roland Lenain, Benoit Thuilot, and Michel Berducat. «Automatic guidance of a four-wheel-steering mobile robot for accurate field operations». In: *Journal of Field Robotics* 26.6-7 (2009), pp. 504–518 (cit. on p. 1).

[3] Spyros Fountas, Nikos Mylonas, Ioannis Malounas, Efthymios Rodias, Christoph Hellmann Santos, and Erik Pekkeriet. «Agricultural robotics for field operations». In: *Sensors* 20.9 (2020), p. 2672 (cit. on p. 1).

[4] Sami Salama Hussen Hajjaj and Khairul Salleh Mohamed Sahari. «Bringing ROS to agriculture automation: hardware abstraction of agriculture machinery». In: *International Journal of Applied Engineering Research* 12.3 (2017), pp. 311–316 (cit. on p. 2).

[5] Ruud Barth et al. «Using ROS for agricultural robotics-design considerations and experiences». In: *Proceedings of the Second International Conference on Robotics and associated High-technologies and equipment for agriculture and forestry.* 2014, pp. 509–518 (cit. on p. 2).

[6] Wageningen University & Research is a collaboration between Wageningen University and the Wageningen Research foundation. *CROPS – Intelligent robot systems for greenhouse horticulture.* 2022. URL: www.crops-robots.eu/ (visited on 11/27/2022) (cit. on p. 2).

[7] Inc. The MathWorks. *xPC Target, For Use with Real-Time Workshop.* 2022. URL: http://www.bmed.mcgill.ca/reklab/archive/xPC/xPC_documentation/xpc_target_ug[1].pdf (visited on 11/27/2022) (cit. on p. 2).

[8] Inc. The MathWorks. *Robotic System Toolbox.* 2022. URL: https://it.mathworks.com/products/ros.html (visited on 11/27/2022) (cit. on p. 2).

[9] M. Galli, R. Barber, S. Garrido, and L. Moreno. «Path planning using Matlab-ROS integration applied to mobile robots». In: *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. 2017, pp. 98–103. DOI: `10.1109/ICARSC.2017.7964059` (cit. on pp. 2, 29).

[10] Inc. Open Source Robotics Foundation. *What is TurtleBot?* 2022. URL: `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#notices` (visited on 11/23/2022) (cit. on pp. 3, 26).

[11] Hoang Thien Nguyen Hong Thai Trinh Thi Khanh Ly and Le Quoc Dzung. «Trajectory Tracking Control for Differential-Drive Mobile Robot by a Variable Parameter PID Controller». In: *International Journal of Mechanical Engineering and Robotics Research* 11 (Aug. 2022), pp. 614–621 (cit. on pp. 3, 12, 35).

[12] Amin Abbasi and Ata Jahangir Moshayedi. «Trajectory Tracking of Two-Wheeled Mobile Robots, Using LQR Optimal Control Method, Based On Computational Model of KHEPERA IV». In: 3 (Mar. 2017), pp. 42–47 (cit. on pp. 3, 14).

[13] Steven M. LaValle. *Planning Algorithms, 726-29*. New York: Cambridge University Press, 2006 (cit. on p. 5).

[14] Spyros G. Tzafestas. *Introduction to Mobile Robot Control, 41-43*. London: Elsevier Insights, 2014 (cit. on p. 7).

[15] BN Biswas, Somnath Chatterjee, SP Mukherjee, and Subhradeep Pal. «A discussion on Euler method: A review». In: *Electronic Journal of Mathematical Analysis and Applications* 1.2 (2013), pp. 2090–2792 (cit. on p. 9).

[16] Héctor J Sussmann and Velimir Jurdjevic. «Controllability of nonlinear systems». In: *Journal of Differential Equations* 12.1 (1972), pp. 95–116. ISSN: 0022-0396. DOI: `https://doi.org/10.1016/0022-0396(72)90007-1`. URL: `https://www.sciencedirect.com/science/article/pii/0022039672900071` (cit. on p. 9).

[17] G. Klancar, D. Matko, and S. Blazic. «Mobile Robot Control on a Reference Path». In: *Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation Intelligent Control, 2005.* 2005, pp. 1343–1348. DOI: `10.1109/.2005.1467211` (cit. on pp. 9, 11, 34).

[18] N. Leenaa and K. K. Saju. «Modelling and trajectory tracking of wheeled mobile robots». In: *Procedia Technology* 24 (2016), pp. 538–545 (cit. on p. 10).

[19] B. D. Hirpo and W. Zhongmin. «Design and control for differential drive mobile». In: *International Journal of Engineering Research & Technology* 6 (2017), pp. 327–334 (cit. on p. 10).

[20] et al. L. Kinam. «Design of Fuzzy-PID controller for path tracking of mobile robot with differential drive». In: *International Journal of Fuzzy Logic and Intelligent Systems* 18 (2018), pp. 220–228 (cit. on p. 10).

[21] et al. P. K. Padhy. «Modeling and position control of mobile robot». In: *Proc. 11th IEEE International Workshop on Advanced Motion Control* (2010), pp. 100–105 (cit. on pp. 10, 12).

[22] Rames C Panda. *Introduction to PID controllers: theory, tuning and application to frontier areas.* BoD–Books on Demand, 2012 (cit. on p. 10).

[23] Filippo Arbinolo. «Modelling and Control of a Skid-Steering Mobile Robot for Indoor Trajectory Tracking Applications». In: *http://webthesis.biblio.polito.it/id/eprint/14632* (2020), p. 186 (cit. on pp. 10, 13).

[24] Ibrahim A El-Sharif, Fathi O Hareb, and Amer R Zerek. «Design of discrete-time PID controller». In: *International Conference on Control, Engineering & Information Technology (CEIT'14).,(hal. 110-115).* 2014 (cit. on p. 12).

[25] Luca Zaccarian and Andrew R Teel. *Modern anti-windup synthesis: control augmentation for actuator saturation.* Vol. 36. Princeton University Press, 2011 (cit. on p. 13).

[26] Ahmad Taher Azar and Fernando E Serrano. «Design and modeling of anti wind up PID controllers». In: *Complex system modelling and control through intelligent soft computations.* Springer, 2015, pp. 1–44 (cit. on p. 13).

[27] Locatelli and Sieniutycz. «Optimal Control: An Introduction». In: *Applied Mechanics Reviews* 55.3 (June 2002), B48–B49 (cit. on p. 14).

[28] Stephen Boyd. *Continuous time linear quadratic regulator.* Stanford University, 2009 (cit. on p. 15).

[29] Open Robotics. *Getting started with ROS.* 2022. URL: `http://wiki.ros.org/it` (visited on 11/24/2022) (cit. on p. 18).

[30] M. Quigley, B. Gerkey, and W.D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System.* O'Reilly Media, 2015. ISBN: 9781449325510. URL: `https://books.google.it/books?id=Hnz5CgAAQBAJ` (cit. on p. 18).

[31] The Construct. *The Beginner's Guide to ROS.* 2022. URL: `https://www.theconstructsim.com/about-ros-robot-operating-system/` (visited on 11/24/2022) (cit. on p. 18).

[32] Zandra B Rivera, Marco C De Simone, and Domenico Guida. «Unmanned ground vehicle modelling in Gazebo/ROS-based environments». In: *Machines* 7.2 (2019), p. 42 (cit. on p. 19).

[33] Omar Elmofty. *ROS2 vs. ROS1— key differences and which one is better?* 2022. URL: `https://medium.com/@oelmofty/ros2-how-is-it-better-than-ros1-881632e1979a` (visited on 11/24/2022) (cit. on p. 19).

[34] Open Robotics. *ROS Computation Graph Level.* 2022. URL: `http://wiki.ros.org/ROS/Concepts` (visited on 11/24/2022) (cit. on p. 20).

[35] Open Robotics. *About Gazebo.* 2022. URL: `https://gazebosim.org/about` (visited on 11/24/2022) (cit. on p. 22).

[36] Open Robotics. *Ubuntu install of ROS Noetic.* 2022. URL: `http://wiki.ros.org/noetic/Installation/Ubuntu` (visited on 11/24/2022) (cit. on p. 23).

[37] Matthew Elwin. *Catkin and Packages.* 2022. URL: `https://nu-msr.github.io/me495_site/lecture02_catkin.html` (visited on 11/24/2022) (cit. on p. 23).

[38] Open Robotics. *Creating a catkin Package.* 2022. URL: `http://wiki.ros.org/ROS/Tutorials/CreatingPackage` (visited on 11/24/2022) (cit. on p. 24).

[39] Open Robotics. *Understanding ROS Nodes.* 2022. URL: `http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes` (visited on 11/24/2022) (cit. on p. 24).

[40] Open Robotics. *Writing a Simple Publisher and Subscriber (C++).* 2022. URL: `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber` (visited on 11/24/2022) (cit. on p. 25).

[41] Open Robotics. *Writing a Simple Publisher and Subscriber (Python).* 2022. URL: `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber` (visited on 11/24/2022) (cit. on p. 25).

[42] Inc. Open Source Robotics Foundation. *Waffle robot.* 2022. URL: `https://www.turtlebot.com/` (visited on 11/24/2022) (cit. on p. 26).

[43] Inc. The MathWorks. *ROS Toolbox.* 2022. URL: `https://it.mathworks.com/help/ros` (visited on 11/24/2022) (cit. on p. 30).

[44] Gregor Klančar, Drago Matko, and Sašo Blažič. «Mobile Robot Control on a Reference Path». In: *Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation Intelligent Control, 2005.* (2005), pp. 1343–1348 (cit. on p. 35).

[45] Amin Abbasi and Ata Jahangir Moshayedi. «Trajectory Tracking of Two-Wheeled Mobile Robots, Using LQR Optimal Control Method, Based On Computational Model of KHEPERA IV». In: *Journal of Simulation and Analysis of Novel Technologies in Mechanical Engineering* 10 (2018), pp. 41–50 (cit. on p. 35).

[46] Wen Tan, Jizhen Liu, Tongwen Chen, and Horacio J Marquez. «Comparison of some well-known PID tuning formulas». In: *Computers & chemical engineering* 30.9 (2006), pp. 1416–1423 (cit. on p. 40).

[47] Daniele Masti, Mario Zanon, and Alberto Bemporad. «Tuning LQR controllers: A sensitivity-based approach». In: *IEEE Control Systems Letters* 6 (2021), pp. 932–937 (cit. on p. 40).