الجمـــهوريـــة الجـــزائريـــة الديـمقـراطيـــــة الشـــعبيـــــة
THE PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
وزارة التعـلـيـم الـعـالـــي والـبـحـث الـعـلـمـي
THE MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
جـامـعـة عمّـــار ثـلـيـــجـــي بالأغـــواط
AMAR TELIDJI UNIVERSITY OF LAGHOUAT

كـلـيـــــة الـتكنولوجيا
FACULTY OF TECHNOLOGY
قسم الالكتروتقني
DEPARTMENT OF ELECTROTECHNIC

Master's thesis

Domain :	Sciences and Technologies
Field :	Automatic
Option :	Automatic and systems

Theme

Control of wheeled mobile robot using ROS

Presented by:
Mr. Abdelatif GHOBSI
Mr. Zakaria BELGHIT

Publicly supported ahead of a jury composed of :

Mrs. Fatima CHOUIREB	*Prof.*	*Chairwoman*
Mr. Mohammed BELKHEIRI	*Prof.*	*Examiner*
Mr. Belkacem RAHMANI	*M.C.B*	*Supervisor*

Academic year 2021/2022

DEDICATION

This modest work would not be possible without the help of **Allah** all mighty, and like the words of his prophet PBUH, he who does not thank the people is not thankful to **Allah**, and for that, this modest work is dedicated to :

My Beloved parents, **Ghobsi Mohammed** and **Marfoua Fadila**, who have supplied me with a bottomless well of inspiration, love and strength throughout the entirety of my life.

My two brothers, **Abdelkader** and **Khaled**, who've supported me through thick and thin and shown interest in my work as well as some thought full guidance and advice throughout my journey.

My friends, **Bedj Ali** and **Seghier Aissa**, thank you.

As well as every friend, family member, student or teacher that i had the honor of crossing paths with, Thank you.

Abdelatif GHOBSI.

DEDICATION

First, I would like to thank **Allah** Almighty for giving me patience and strength to finish this work.

I dedicate this modest work : To my dear and beloved mother and father who have paved the path that i walked on and done everything for me to succeed in my life and wished nothing but the best for me, may Allah protect them.

To my soft-hearted brothers as well as my admirable sisters who have shown me support and filled me with encouragements during these years of study.

A relentless partner with an inspiring work ethic, **Abdelatif GHOBSI**.

Every student, classmate and teacher that i have had the honour of working with or learning under, Thank you.

Zakaria BELGHIT.

ACKNOWLEDGMENTS

With the help of Almighty God, we were able to accomplish this work, which was carried out within the Telecommunication, Signals and Systems Laboratory **TSSL**, and the electrical engineering department at Amar Telidji University of Laghouat.

At the end of this work, We would like to thank our supervisor Mr. **Belkacem RAHMANI** for his enormous follow-up, for his constant attention to our work, for his wise advice and his listening, which were essential for the success of this dissertation, and for his confidence.
We would like to thank Mr. **Khalil OBATI** who helped us a lot.

We also thank all of those who contributed from near or far to the realization of this work, who will find here our expression of our full gratitude and appreciation.

We would also like to thank the members of the jury for accepting to examine our work. Thank you for the constructive remarks, and for your relevant comments.

We would like to express our sincere thanks to the teachers who have guided us throughout this academic journey.

ABSTRACT

The Robot Operating System (ROS) is an open-source framework that allows robot developers to create robust software for a wide variety of robot platforms, sensors, and effectors.

The subject of this thesis is the motion control of the wheeled mobile robot P3-DX using ROS. With reference to the kinematic model, we review several control strategies for the trajectory tracking problem. Specifically, a control based on approximate linearization, nonlinear control, and dynamic feedback linearization control. Where, the desired trajectory is carried out by the planner that uses the cubic cartesian polynomials method.

Our objective encompassed the integration of ROS and the p3dx robot in order to implement the control laws for simulation and application. P2OS packages were utilized for simulation on the Gazebo environment, and the RosAria node was used as the main driver for the robot. Both the simulation and experimental results agree very well and show the usability of ROS.

Keywords: wheeled mobile robots, ROS, path planning, trajectory tracking, linear control, nonlinear control, dynamic feedback linearization control, P3-Dx, P2OS, ROSARIA.

ملخص

نظام تشغيل الروبوت (ROS) هو إطار مفتوح المصدر يسمح لمطوري الروبوتات بإنشاء برامج قوية لمجموعة متنوعة من منصات الروبوت وأجهزة الاستشعار والمؤثرات. موضوع هذه الأطروحة هو التحكم في حركة الروبوت المحمول ذي العجلات P3-DX باستخدام ROS. بالإشارة إلى النموذج الحركي، نستعرض العديد من استراتيجيات التحكم لمشكلة تتبع المسار. على وجه التحديد، التحكم القائم على التحكم الخطي التقريبي، والتحكم غير الخطي، والتحكم الديناميكي في التغذية الراجعة. حيث يتم تنفيذ المسار المطلوب بواسطة المخطط الذي يستخدم طريقة متعددة الحدود الديكارتية المكعبة. شمل هدفنا تكامل ROS والروبوت p3dx من أجل تنفيذ قوانين التحكم للمحاكاة والتطبيق. واستخدمت P2OS للمحاكاة في بيئة GAZEBO، واستخدمت عقدة ROSARIA كمحرك رئيسي للروبوت. تتفق كل من المحاكاة والنتائج التجريبية بشكل جيد للغاية وتظهر قابلية استخدام ROS.

الكلمات الرئيسية: الروبوتات المحمولة ذات العجلات، ROS، تخطيط المسار، تتبع المسار، التحكم القائم على التحكم الخطي التقريبي، والتحكم غير الخطي، والتحكم الديناميكي في التغذية الراجعة، P2OS، P3-DX، ROSARIA.

Résumé

Robot Operating System (ROS) est un framework open-source qui permet aux développeurs de robots de créer des logiciels robustes pour une grande variété de plateformes de robots, de capteurs et d'effecteurs.

Le sujet de cette thèse est la commande de mouvement du robot mobile à roues P3-DX utilisant ROS. En ce qui concerne le modèle cinématique, nous examinons plusieurs stratégies de commande pour le problème de suivi de trajectoire. Plus précisément, une commande basée sur la linéarisation approximative, la commande non linéaire et la commande de linéarisation par un retour d'état dynamique. Où la trajectoire souhaitée est effectuée par le planificateur qui utilise la méthode des polynômes cartésiens cubiques. Notre objectif comprenait l'intégration de ROS et du robot p3dx afin de mettre en œuvre les lois de commande pour la simulation et l'application. Les paquets P2OS ont été utilisés pour la simulation sur l'environnement Gazebo, et le nœud RosAria a été utilisé comme pilote principal pour le robot. La simulation et les résultats expérimentaux concordent très bien et montrent la facilité d'utilisation de ROS.

Mots clés: robots mobiles à roues, ROS, planification de trajectoire, suivi de trajectoire, commande linéaire, commande non linéaire, commande de linéarisation par un retour d'état dynamique., P3-Dx, P2OS, ROSARIA.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

WMR	Wheeled Mobile Robot.
DDMR	The Differential Drive Wheeled Mobile Robot.
DFL	Dynamic Feedback Linearization.
ROS	Robot Operating System.
URDF	The unified robot description format.
TF	Coordinate Frames and Transforms.
OSRF	Open Source Robotics Foundation.
BSD	Berkeley Source Distribution.

GENERAL INTRODUCTION

It is at least two decades since the conventional robotic manipulators have become a common manufacturing tool for different industries, from automotive to pharmaceutical. The proven benefits of utilizing robotic manipulators for manufacturing in different industries motivated scientists and researchers to try to extend the applications of robots to many other areas. To extend the application of robotics, scientists had to invent several new types of robots other than conventional manipulators. The new types of robots can be categorized in two groups : redundant (and hyper-redundant) manipulators and mobile (ground, marine, and aerial) robots. These two groups of robots have more freedom for their mobility, which allows them to do tasks that the conventional manipulators cannot do. [4]

Research related to design, modeling and control of mobile robots has been one of the most active areas of robotics in the last few decades. The use of mobile robots in various areas of human life has been continuously increasing. Mobile robots are being used as autonomous cleaning devices at homes, for factory automation, to patrol national borders, for bomb disposals, for hazardous waste cleaning, etc. One of the goals of research in mobile robotics is to make the mobile robots completely autonomous, that is, the robots are capable of making decisions based on sensing the environment they are in. Another area where mobile robots are employed is in cooperative control of multiple mobile robots which has received considerable attention in the last two decades with application in distributed transportation, monitoring and multi-point surveillance. [17]

A mobile robot must have a capable sensor suite and the ability to process the data from the sensor, estimate its location, construct a map, identify an optimal and feasible path from its current location to the goal, and maneuver to the objective. This requires a robust software framework to allow the robot platform, controllers, and sensors to work harmoniously in order to achieve the objective. Due to this we exploite Robot Operating System (ROS), which is becoming a widely popular method for writing robot software, primarily because of its flexibility, robustness, and modularity. Additionally, ROS is completely open-source, creating an environment in which the spread of knowledge and learning is prevalent within the robotics community. Because of these qualities, each modular part of ROS has a plug-and-play feel, allowing users,developers, and researchers to pick whichever packages are best for their robots and the ability to configure them in a simple manner.

As part of our work, we are interested in the study of the wheeled mobile robot P3-DX provided by the Telecommunication, Signals and Systems Laboratory, at the University of

Laghouat. The objective is to control this robot using ROS. The contributions of this work are summarized below :

In chapter 1. Primary, the kinematic model of the differential drive robot under the non-holonomic constraint is described. In addition, for the trajectory planning problem the planner that uses the cubic Cartesian polynomials method is explained. Finally, differentiable control laws were presented, namely dynamic feedback linearisation, non-linear and linear controls.

In chapter 2, the fundamental concepts of ROS are highlighted. Gazebo simulator, robot hardware, and the necessary packages to integrate ROS with the P3-DX robot are also given and discussed. Chapter 3 presents simulation and experimental results.

Finally, a conclusion that summarizes the work and some possible future plans finish the manuscript.

CHAPTER 1

Modelling, Planning, and Motion Control

1.1 Introduction

The differential drive wheeled mobile robot (DDMR) is ubiquitous in the robotics market. Nearly every budding roboticist has developed such a platform since they are quite easily constructed. Surprisingly then, considering its popularity, the platform is non-holonomic and its control behavior is non-linear. A great many control schemes have been developed to address the problem, using several approaches. Among them, linearization around an operating point, dynamic feedback linearization (DFL), and non-linear algorithms bounded to stable regions by Lyapunov functions. [16]

In this first chapter, we will describe the kinematic model for wheeled mobile robots (WMRs) under the non-holonomic constraint of pure rolling and non-slipping. we will also explore the trajectory planning via Cartesian polynomials method. and finally, based on the kinematic model, we will present differentiable, time-varying kinematic controllers for the tracking control problems.

1.2 Mathematical model

Following [3], this section formulates the kinematic model of a WMR in Cartesian coordinates under non-holonomic constraints.

1.2.1 Non-Holonomic Constraint

Wheeled vehicles are generally subjected to a constraint. For instance, a car can reach any final configuration in its plane, but it can never move sideways. Hence, depending on the goal configuration, it requires to perform a series of maneuvers (such as parallel parking) to reach the desired state.

First, holonomic and non-holonomic systems have to be defined. Let's consider a mechanical system with generalized coordinates $q \in C$, where C is the configuration space of the proposed system and coincides with $\mathbb{R}^n$. For such system, a constraint is called Kinematic when it only involves generalized coordinates (q) and velocities ($\dot{q}$).

Kancrantic Constraints are usually defined in Pfaffian Form

$$v_i^T(q)\dot{q} = 0 \quad i = 1, \ldots, k < n \tag{1.1}$$

where v_i's are k linearly independent vectors.

If all of the kinematic constraints defined by Equation 1.1 are integrable to a form of

$$h_i(q) = m_i \quad i = 1, \ldots, k < n$$

where, m_i is the integration constant, then they are considered to be holonomic constraints and the system subjected to them is called a holonomic system. Joints in a robotic manipulator are common example of such constraints.

Each holonomic constraint causes a loss of accessibility of the system in its configuration space. Hence, for a system with k holonomic constraints, the accessible configurations are reduced to a $(n-k)$ dimensional subset of C.

A non-holonomic system on the other hand, is subjected to at least one nonintegrable (i.e. non-holonomic) constraint. Although such constraint limits the local mobility of the system, due to its non-integrable nature, the accessibility to C is not affected. Hence, generalized coordinates are not reduced. However, generalized velocities in a system subjected to k non-holonomic constraint belongs to a $(n-k)$ dimensional subspace.

Wheels are typical sources of non-holonomic constraints. Consider the disk in Figure 1.1 with generalized coordinates $q = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$, assuming the disk can only roll on the touching plane without slipping to the sides (i.e. there is no velocity component for the contact point perpendicular to the plane containing the disk).

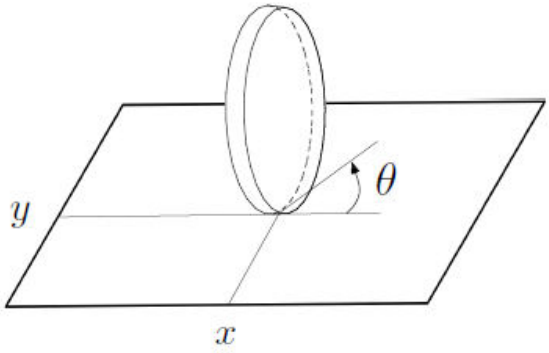

FIGURE 1.1 Pure rolling disk and its generalized coordinates in 2D plane.

This can be defined as :

$$\dot{x}\sin\theta - \dot{y}\cos\theta = 0 \tag{1.2}$$

Rewriting Equation (1.2) in pfaffian form will result in

$$[\sin\theta \quad -\cos\theta \quad 0]\dot{q} = 0 \tag{1.3}$$

As it can be seen, Equation (1.3) is not integrable causing the nature of the wheel to be non-holonomic. Also, it should be emphasized that this constraint implies no loss in accessibility of the wheel configuration space, meaning that wheel can reach any goal configuration $q_f = \begin{bmatrix} x_f & y_f & \theta_f \end{bmatrix}^T$ starting from any initial state $q_i = \begin{bmatrix} x_i & y_i & \theta_i \end{bmatrix}^T$.

1.2.2 Robot Kinematics

Reordering k kinematic constraints in Equation 1.1 into matrix form $V^T(q)\dot{q} = 0$, shows that the generalized velocities $(\dot{q})$ belongs to null space of $V^T(q)$, which is $(n-k)$ dimensional and agrees with what was stated earlier in this chapter.

Choosing a basis for $\mathcal{N}\left(V^T(q)\right)$ denoted by $[b_1(q)\dots b_{n-k}(q)]$ a kinematic model of the constrained mechanical system is given by :

$$\dot{q} = \sum_{i=1}^{n-k} b_i(q)u_i = B(q)\mathbf{u} \tag{1.4}$$

where $\mathbf{u} = [u_1 \dots u_{n-k}]^T \in \mathbb{R}^{n-k}$ is the input vector and $q \in \mathbb{R}^n$ is the state vector.

The basis for nullspace of $V^T(q)$ is not unique and typically, it can be chosen such that inputs u_i represent a physical concept. However, these inputs should not directly represent forces or torques, hence the name kinematic model.

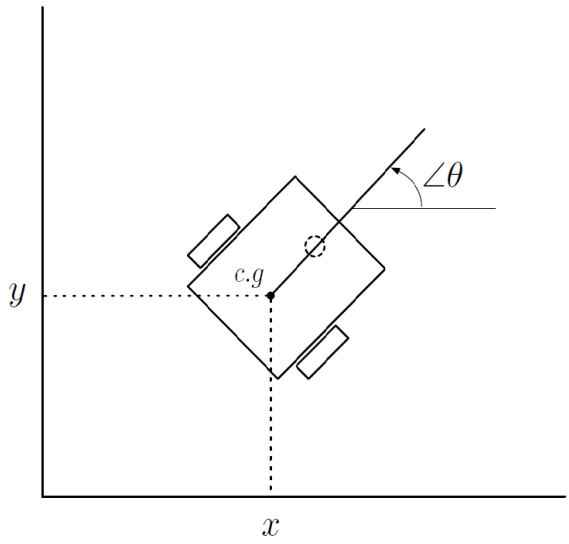

FIGURE 1.2 Generalized coordinates for a mobile robot.

Consider the mobile robot in Figure 1.2. Using generalized coordinate vector $q = \begin{bmatrix} x & y & \theta \end{bmatrix}$ the robot's posture can be defined on its whole configuration space.
The wheels driving the robot make it non-holonomic and imposes the pure rolling constraint on the system which as discussed before, is expressed as

$$V^T(q)\dot{q} = \begin{bmatrix} \sin\theta & -\cos\theta & 0 \end{bmatrix} \dot{q} = 0 \tag{1.5}$$

a basis for $\mathcal{N}\left(V^T(q)\right)$ is then chosen as

$$B(q) = [b_1(q) \quad b_2(q)] = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \tag{1.6}$$

Using this basis and based on Equation (1.4) the kinematic model will be

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{1.7}$$

where, the inputs have clear physical interpretation, v and ω are the linear velocity and

angular velocity of the robot, respectively, as shown in Figure 1.2.

There exists a one to one relation between formerly mentioned velocities and actual velocity inputs, which are angular speed of two wheels denoted by ω_L and ω_R for left and right wheels, respectively and is governed by :

$$v = \frac{r\left(\omega_R + \omega_L\right)}{2} \quad \omega = \frac{r\left(\omega_R - \omega_L\right)}{l} \tag{1.8}$$

where, r is the radius of the wheels and l is the distance between the wheels as shown in Figure 1.3

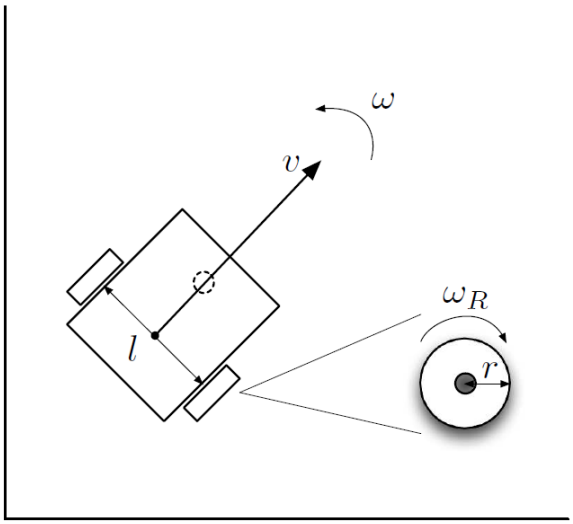

FIGURE 1.3 Linear and Angular velocity of the robot.

1.3 Main problems

The robotics and control communities are trying to answer some of the big questions related to mobile robots. A WMR must move from its origin to its final destination, satisfying speed and/or position constraints along the way.

The task is broken down into different problems to be solved individually or collectively. These problems are divided as shown in Figure 1.4 :

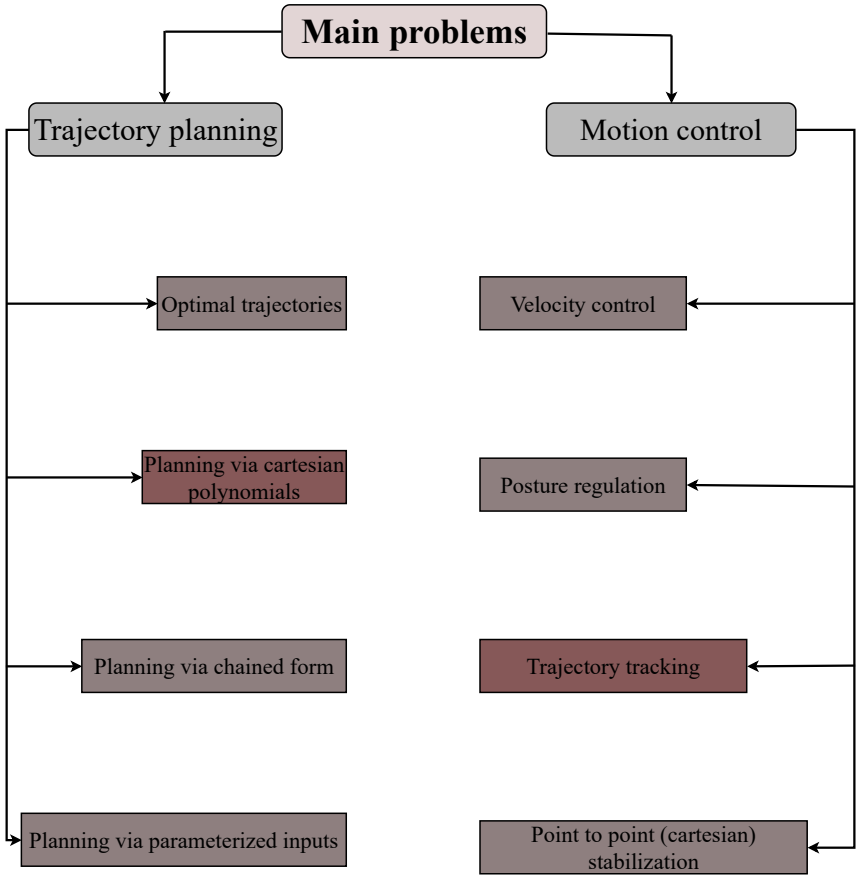

FIGURE 1.4 Problems diagram.

In this work, we have chosen to use planning via Cartesian polynomials, and we have focused on the trajectory tracking problem.

1.4 Trajectory planning

According to [23], and [3]. Mobile robots need a trajectory to track and reach their targets. Planning this trajectory can be done in a variety of ways to satisfy conditions such as minimum distance, minimum travel time, and so on. However, in general, this task can be broken down into finding paths and defining the required timing law along those paths.

1.4.1 Path and timing law

Consider a trajectory $q(t)$, $t \in [t_i, t_f]$ that guides a WMR from initial configuration $q(t_i) = q_i$ to final configuration $q(t_f) = q_f$ in time $T = t_i - t_f$. This trajectory can be broken down into a geometric path $q(s)$, where $\frac{dq(s)}{ds} \neq 0$ and a timing law $s = s(t)$ where $s(t)$ is monotonically increasing function of time on $[t_i, t_f]$,i.e $\dot{q}(t) \geq 0$. Generalized velocity vector can then be obtained as :

$$\dot{q}(t) = \frac{dq}{dt} = \frac{dq}{ds}\frac{ds}{dt} = q'\dot{s} \tag{1.9}$$

where q' is the tangent vector to the path.

1.4.2 Effects of kinematic constraint

A kinematic constraint such as (1.5) can be expressed as :

$$A(q)\dot{q} = A(q)q'\dot{s} = 0 \tag{1.10}$$

if $s(t)$ os strictly increasing,i.e. $\dot{s} > 0$, then it is trivial that :

$$A(q)q' = 0 \tag{1.11}$$

has to hold. Essentially it means that in a mechanical system subject to non-holonomic constraints a geometric path is admissible if and only if it satisfies (1.10). A set of all admissible paths can be derived as a solution to :

$$q' = \sum_{i=1}^{n-k} b_i(q)\hat{u}_i = B(q)\hat{\mathbf{u}} \tag{1.12}$$

where, $\hat{u}$ is the vector of geometric inputs related to kinematic inputs vector u by $u(t) = \hat{u}(s)\dot{s}(t)$. In order to acquire a unique admissible path, selecting the geometric inputs for

$s \in [s_i, s_f]$ would suffice. In the case of non-holonomic robot, admissible paths must satisfy :

$$\begin{bmatrix} sin\theta & -cos\theta & 0 \end{bmatrix} q' = 0 \tag{1.13}$$

Therefore, all the admissible paths can be formulated as :

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{v} \\ \hat{\omega} \end{bmatrix} \tag{1.14}$$

where :

$$\begin{aligned} v(t) &= \hat{v}(s)\dot{s}(t) \\ \omega(t) &= \hat{\omega}(s)\dot{s}(t) \end{aligned} \tag{1.15}$$

The kinematic constraint in equation (1.13) states that an admissible path for a non-holonomic robot should have a tangent aligned with the robot's sagittal axis. In other words, no edges or sharp points are allowed on the path.

1.4.3 Differential Flatness

Consider a non-linear system defined by :

$$\begin{aligned} \dot{X} &= F(X) + G(X)u \\ Y &= H(X) + D(X)u \end{aligned} \tag{1.16}$$

Such system is differentially flat if there exists a set of outputs y, where states x and control inputs u can be expressed as unique functions of y and its derivatives :

$$\begin{aligned} X &= F_1\left(Y, \dot{Y}, \ddot{Y}, \ldots, Y^{(n)}\right) \\ U &= F_2\left(Y, \dot{Y}, \ddot{Y}, \ldots, Y^{(n)}\right) \end{aligned} \tag{1.17}$$

Outputs Y are called flat outputs. Cartesian coordinates $[x, y]$ in mobile robots are considered flat outputs, consider geometric model in Equation (1.14), by defining an output Cartesian path $[x(s), y(s)]$ one can calculate the orientation from :

$$\theta(s) = \operatorname{atan} 2\left(y'(s), x'(s)\right) + k\pi \quad k = 0, 1 \tag{1.18}$$

where, k defines if the robot is moving forward ($k = 0$) or backward ($k = 1$) and "atan2" is a variation of arctangent, that calculates the angle between the x axis and the line passing through point (x, y) from origin. The states are then obtained as $q(s) = [x(s), y(s), \theta(s)]^T$ and the geometric velocity inputs are uniquely defined by Equation (1.19).

$$\begin{aligned} \hat{v}(s) &= \pm\sqrt{x'(s)^2 + y'(s)^2} \\ \hat{\omega}(s) &= \frac{y''(s)x'(s) - x''(s)y'(s)}{x'(s)^2 + y'(s)^2} \end{aligned} \tag{1.19}$$

This means that a unique path along with unique velocities can be defined for the robot.

1.4.4 Planning via Cartesian polynomials

As mentioned in [23]. Whenever a WMR admits a set of flat outputs y, it can be used to efficiently solve planning problems. In fact, we can plan the path of y using any interpolation scheme so that the appropriate boundary conditions are met. We can then algebraically calculate the evolution of the other configuration variables from $y(s)$ along with the relevant control inputs. The resulting configuration space path will automatically satisfy the non-holonomic constraints in Equation (1.10).

In particular, consider the problem of planning a path that leads a WMR from an initial configuration $q(s_i) = q_i = [x_i, y_i, \theta_i]^T$ to a final configuration $q(s_f) = q_f = [x_f, y_f, \theta_f]^T$.

As mentioned above, the problem can be solved by interpolating the initial values x_i, y_i and the final values x_f, y_f of the flat outputs x, y. Letting $s_i = 0$ and $s_f = 1$, we can use the following cubic polynomials :

$$\begin{aligned} x(s) &= s^3 x_f - (s-1)^3 x_i + \alpha_x s^2(s-1) + \beta_x s(s-1)^2 \\ y(s) &= s^3 y_f - (s-1)^3 y_i + \alpha_y s^2(s-1) + \beta_y s(s-1)^2 \end{aligned} \tag{1.20}$$

that automatically satisfy the boundary conditions on x, y. The orientation at each point being related to x', y' by Equation (1.18), it is also necessary to impose the additional boundary conditions

$$\begin{aligned} x'(0) &= k_i \cos\theta_i \quad x'(1) = k_f \cos\theta_f \\ y'(0) &= k_i \sin\theta_i \quad y'(1) = k_f \sin\theta_f \end{aligned} \tag{1.21}$$

where $k_i \neq 0$,$k_f \neq 0$ are free parameters that must however have the same sign. This condition is necessary to guarantee that the WMR arrives in q_f with the same kind of motion (forward or backward) with which it leaves q_i, in fact, since $x(s)$ and $y(s)$ are cubic

polynomials, the Cartesian path does not contain motion inversions in general.

For example, by letting $k_i = k_f = k > 0$, one obtains

$$\begin{bmatrix} \alpha_x \\ \alpha_y \end{bmatrix} = \begin{bmatrix} k\cos\theta_f - 3x_f \\ k\sin\theta_f - 3y_f \end{bmatrix} \quad \begin{bmatrix} \beta_x \\ \beta_y \end{bmatrix} = \begin{bmatrix} k\cos\theta_i + 3x_i \\ k\sin\theta_i + 3y_i \end{bmatrix} \tag{1.22}$$

The choice of k_i and k_f has a precise influence on the obtained path. In fact, by using Equation (1.19) it is easy to verify that

$$\tilde{v}(0) = k_i \quad \tilde{v}(1) = k_f \tag{1.23}$$

The evolution of the robot orientation along the path and the associated geometric inputs can then be computed by using Equations. (1.18) and (1.19), respectively.

1.5 Motion control

as reported by [15], [23], and [8]. Motion control problems for WMR are generally formulated with reference to the kinematic model (1.7), by assuming that the control inputs directly determines the generalized velocity $\dot{q}$, this means that the inputs are the driving and steering velocity inputs v and ω. There are two main reasons for this simplified assumption. First, it is possible to cancel the dynamic effect via state feedback, so that the control problem is actually transferred to the second-order kinematic model, and from there to the first-order kinematic model. Second, most WMRs like the P3-DX have low-level control loops integrated into their hardware or software architecture, so it is impossible to directly command the wheel torques. These control loops accept a reference value for the angular velocity of the wheel as an input, This reference value is reproduced as accurately as possible by a standard controller (such as a PID controller). In this situation, the actual input that can be used for high levels of control is exactly the reference speed.

From a practical point of view, the most relevant issue is certainly trajectory tracking. This is because mobile robots need to operate invariably in unstructured workspaces with obstacles. Obviously, moving the robot along (or close to) a pre-planned trajectory significantly reduces the risk of collision. [23] this control problem is illustrated in Figure (1.5) :

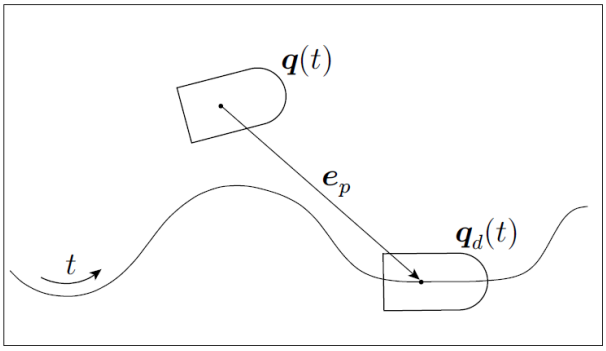

FIGURE 1.5 Trajectory tracking. [23]

the robot must asymptotically track a desired Cartesian trajectory $(x_d(t), y_d(t))$, starting from an initial configuration $q_0 = [x_0, y_0, \theta_0]^T$ that may or may not be 'matched' with the trajectory. For the tracking problem to be soluble, it is necessary that this trajectory $(x_d(t), y_d(t))$ is admissible for the kinematic model (1.7), i.e., it must satisfy the following equations :

$$\begin{aligned} \dot{x}_d &= v_d \cos\theta_d \\ \dot{y}_d &= v_d \sin\theta_d \\ \dot{\theta}_d &= \omega_d \end{aligned} \tag{1.24}$$

the orientation along the desired trajectory (xd(t), yd(t)) can be computed as :

$$\theta_d(t) = \text{Atan}\,2\left(\dot{y}_d(t), \dot{x}_d(t)\right) + k\pi \quad k = 0, 1 \tag{1.25}$$

as well as the reference inputs :

$$\begin{aligned} v_d(t) &= \pm\sqrt{\dot{x}_d^2(t) + \dot{y}_d^2(t)} \\ \omega_d(t) &= \frac{\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)}{\dot{x}_d^2(t) + \dot{y}_d^2(t)} \end{aligned} \tag{1.26}$$

Note that (1.25) and (1.26), correspond respectively to (1.18) and (1.19) with $s = t$.

By comparing the desired state $q_d(t) = [x_d(t), y_d(t), \theta_d(t)]^T$ with the current measured state $q(t) = [x(t), y(t), \theta_d(t)]^T$ it is possible to compute an error vector that can be fed to the controller. However, rather than using directly the difference between q_d and q, it is

convenient to define the tracking error as :

$$e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d - x \\ y_d - y \\ \theta_d - \theta \end{bmatrix} \tag{1.27}$$

The positional part of e is the Cartesian error $e_p = \begin{bmatrix} x_d - x & y_d - y \end{bmatrix}^T$ expressed in a reference frame aligned with the current orientation θ of the robot (see Figure (1.5)). By differentiating e with respect to time, and using Equations (1.7) and (1.24), one easily finds :

$$\begin{aligned} \dot{e}_1 &= v_d \cos e_3 - v + e_2\omega \\ \dot{e}_2 &= v_d \sin e_3 - e_1\omega \\ \dot{e}_3 &= \omega_d - \omega. \end{aligned} \tag{1.28}$$

Using the input transformation

$$\begin{aligned} v &= v_d \cos e_3 - u_1 \\ \omega &= \omega_d - u_2 \end{aligned} \tag{1.29}$$

which is clearly invertible, the following expression is obtained for the tracking error dynamics :

$$\dot{e} = \begin{bmatrix} 0 & \omega_d & 0 \\ -\omega_d & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} e + \begin{bmatrix} 0 \\ \sin e_3 \\ 0 \end{bmatrix} v_d + \begin{bmatrix} 1 & -e_2 \\ 0 & e_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1.30}$$

The first term of this dynamics is linear, while the second and third are nonlinear. Moreover, the first and second terms are in general time-varying, due to the presence of the reference inputs $v_d(t)$ and $\omega_d(t)$. [23]

1.5.1 Control based on approximate linearization

The simplest approach to designing a tracking controller consists of using the approximate linearization of the error dynamics around the reference trajectory, on which clearly $e = 0$. This approximation, whose accuracy increases as the tracking error e decreases, is obtained from (1.30) simply setting $sin(e_3) = e_3$ and evaluating the input matrix on the trajectory.

The result is :

$$\dot{e} = \begin{bmatrix} 0 & \omega_d & 0 \\ -\omega_d & 0 & v_d \\ 0 & 0 & 0 \end{bmatrix} e + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1.31}$$

Note that the approximate system is still time-varying. Consider now the linear feedback :

$$\begin{aligned} u_1 &= -k_1 e_1 \\ u_2 &= -k_2 e_2 - k_3 e_3 \end{aligned} \tag{1.32}$$

that leads to the following closed-loop linearized dynamics :

$$\dot{e} = A(t)e = \begin{bmatrix} -k_1 & \omega_d & 0 \\ -\omega_d & 0 & v_d \\ 0 & -k_2 & -k_3 \end{bmatrix} e. \tag{1.33}$$

The characteristic polynomial of matrix A is :

$$p(\lambda) = \lambda\left(\lambda + k_1\right)\left(\lambda + k_3\right) + \omega_d^2\left(\lambda + k_3\right) + v_d k_2\left(\lambda + k_1\right) \tag{1.34}$$

At this point, it is sufficient to let :

$$k_1 = k_3 = 2\zeta a \quad k_2 = \frac{a^2 - \omega_d^2}{v_d} \tag{1.35}$$

with $\zeta \in (0, 1)$ and $a > 0$, to obtain :

$$p(\lambda) = (\lambda + 2\zeta a)\left(\lambda^2 + 2\zeta a\lambda + a^2\right) \tag{1.36}$$

It should be emphasized that, even if the closed-loop eigenvalues are constant : one real negative eigenvalue in $-2\zeta a$ and a pair of complex eigenvalues with negative real part. this control law does not guarantee the asymptotic stability of the state tracking error e, because the system is still time-varying. [23]

1.5.2 Nonlinear control

Consider again the exact expression (1.30) of the tracking error dynamics, now rewritten for convenience in the 'mixed' form :

$$\begin{aligned} \dot{e}_1 &= e_2\omega + u_1 \\ \dot{e}_2 &= v_d \sin e_3 - e_1\omega \\ \dot{e}_3 &= u_2 \end{aligned} \tag{1.37}$$

and the following nonlinear version of the control law (1.32) :

$$\begin{aligned} u_1 &= -k_1\left(v_d, \omega_d\right) e_1 \\ u_2 &= -k_2 v_d \frac{\sin e_3}{e_3} e_2 - k_3\left(v_d, \omega_d\right) e_3 \end{aligned} \tag{1.38}$$

where $k_1(\cdot,\cdot) > 0$ and $k_3(\cdot,\cdot) > 0$ are bounded functions with bounded derivatives, and $k_2 > 0$ is constant. If the reference inputs v_d and ω_d are also bounded with bounded derivatives, and they do not both converge to zero, the tracking error e converges to zero globally, i.e., for any initial condition. A sketch is now given of the proof of this result. Consider the closed-loop error dynamics

$$\begin{aligned} \dot{e}_1 &= e_2\omega - k_1\left(v_d, \omega_d\right) e_1 \\ \dot{e}_2 &= v_d \sin e_3 - e_1\omega \\ \dot{e}_3 &= -k_2 v_d \frac{\sin e_3}{e_3} e_2 - k_3\left(v_d, \omega_d\right) e_3 \end{aligned} \tag{1.39}$$

and the candidate Lyapunov function

$$V = \frac{k_2}{2}\left(e_1^2 + e_2^2\right) + \frac{e_3^2}{2}, \tag{1.40}$$

whose time derivative along the solutions of the closed-loop system is nonincreasing since

$$\dot{V} = -k_1 k_2 e_1^2 - k_3 e_3^2 \leq 0. \tag{1.41}$$

Thus, $\|e(t)\|$ is bounded, $\dot{V}(t)$ is uniformly continuous, and $V(t)$ tends to some limit value. Using Barbalat lemma, $\dot{V}(t)$ tends to zero. From this and analyzing the system equations, one can show that $\left(v_d^2 + \omega_d^2\right) e_i^2 (i = 1, 2, 3)$ tends to zero so that, from the persistency of the (state) trajectory $q(t)$. The actual velocity inputs v and ω must be computed from u_1 and u_2 using (1.29).

Taking advantage of the previous linear analysis, we can choose the gain functions k_1 and k_2 and the constant gain $\bar{k}_2$ as :

$$k_1\left(v_d(t),\omega_d(t)\right) = k_3\left(v_d(t),\omega_d(t)\right) = 2\zeta\sqrt{\omega_d^2(t) + bv_d^2(t)}, \quad k_2 = b, \tag{1.42}$$

with $b > 0$ and $\zeta \in (0,1)$. [23]

1.5.3 Dynamic feedback linearization

A nonlinear controller for trajectory tracking based on exact dynamic feedback linearization is now designed following [7]. With reference to the general class of non-holonomic driftless systems, the DFL problem consists in finding, if possible, a dynamic state feedback compensator of the form

$$\begin{aligned} \dot{\xi} &= a(q,\xi) + b(q,\xi)u \\ w &= c(q,\xi) + d(q,\xi)u \end{aligned} \tag{1.43}$$

with ν-dimensional state ξ and m-dimensional external input u, such that the closed-loop system (1.7)-(1.43) is equivalent, under a state transformation $z = T(q,\xi)$, to a linear controllable system. [12]

The starting point is the definition of an appropriate m-dimensional system output $\eta = h(q)$, to which a desired behavior can be assigned (in our case, track a desired trajectory). One then proceeds by successively differentiating the output until the input appears in a nonsingular way. At some stage, the addition of integrators on a subset of the input channels may be necessary in order to avoid subsequent differentiation of the original inputs. This dynamic extension algorithm builds up the state ξ of the dynamic compensator (1.43). The algorithm terminates after a finite number of differentiations whenever the system is invertible from the chosen output. If the sum of the output differentiation orders equals the dimension $n + \nu$ of the extended state space, full input-state-output linearization is also obtained. The closed-loop system is then equivalent to a set of decoupled input-output chains of integrators from u_i to $\eta_i (i = 1,\ldots,m)$.

We illustrate this exact linearization procedure for the unicycle model (1.7). Define the linearizing output vector as $\eta = (x, y)$. Differentiation w.r.t. time then yields

$$\dot{\eta} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{1.44}$$

showing that only v affects $\dot{\eta}$, while the angular velocity ω cannot be recovered from this first-order differential information. In order to proceed, we need therefore to add an integrator

(whose state is denoted by ξ) on the linear velocity input

$$v = \xi, \quad \dot{\xi} = a \quad \Longrightarrow \quad \dot{\eta} = \xi \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \tag{1.45}$$

being the new input a the linear acceleration of the unicycle. Differentiating further

$$\ddot{\eta} = \dot{\xi} \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} + \xi\dot{\theta} \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & -\xi\sin\theta \\ \sin\theta & \xi\cos\theta \end{bmatrix} \begin{bmatrix} a \\ \omega \end{bmatrix} \tag{1.46}$$

and the matrix multiplying the modified input (a, ω) is nonsingular provided that $\xi \neq 0$. Under this assumption, we can define

$$\begin{bmatrix} a \\ \omega \end{bmatrix} = \begin{bmatrix} \cos\theta & -\xi\sin\theta \\ \sin\theta & \xi\cos\theta \end{bmatrix}^{-1} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1.47}$$

so as to obtain

$$\ddot{\eta} = \begin{bmatrix} \ddot{\eta}_1 \\ \ddot{\eta}_2 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = u. \tag{1.48}$$

The resulting dynamic compensator is

$$\begin{aligned} \dot{\xi} &= u_1\cos\theta + u_2\sin\theta \\ v &= \xi \\ \omega &= \frac{u_2\cos\theta - u_1\sin\theta}{\xi}. \end{aligned} \tag{1.49}$$

Since the dynamic compensator is one-dimensional, we have $n+\nu = 3+1 = 4$, equal to the total number of output differentiations in Equation (1.48). Therefore, in the new coordinates

$$\begin{aligned} z_1 &= x \\ z_2 &= y \\ z_3 &= \dot{x} = \xi\cos\theta \\ z_4 &= \dot{y} = \xi\sin\theta \end{aligned} \tag{1.50}$$

the extended system is fully linearized in a controllable form and described by the two

chains of second-order input-output integrators given by equatoin (1.48), rewritten as

$$\begin{aligned} \ddot{z}_1 &= u_1 \\ \ddot{z}_2 &= u_2. \end{aligned} \tag{1.51}$$

Note that the DFL controller (1.49) has a potential singularity at $\xi = v = 0$, i.e., when the unicycle is not rolling. The occurrence of such singularity in the dynamic extension process has been shown to be structural for non-holonomic systems [6]. This difficulty must be obviously taken into account when designing control laws on the equivalent linear model.

Assume the robot must follow a smooth output trajectory $(x_d(t), y_d(t))$ which is persistent, i.e., such that the nominal control input $v_d = (\dot{x}_d^2 + \dot{y}_d^2)^{1/2}$ along the trajectory does never go to zero. On the equivalent linear and decoupled system (1.51), it is straightforward to design a globally exponentially stabilizing feedback for the desired trajectory (with linear cartesian transients) as

$$\begin{aligned} u_1 &= \ddot{x}_d(t) + k_{p1}\left(x_d(t) - x\right) + k_{d1}\left(\dot{x}_d(t) - \dot{x}\right) \\ u_2 &= \ddot{y}_d(t) + k_{p2}\left(y_d(t) - y\right) + k_{d2}\left(\dot{y}_d(t) - \dot{y}\right), \end{aligned} \tag{1.52}$$

with PD gains chosen as $k_{pi} > 0, k_{di} > 0$, for $i = 1, 2$.

In the implementation, velocities $\dot{x}$ and $\dot{y}$ can be computed via the last two expressions in Equation (1.50), as a function of the robot state and of the compensator state ξ. Alternatively, one can use estimates of $\dot{x}$ and $\dot{y}$ obtained from odometric measurements. This solution is more robust with respect to unmodeled dynamics.

We conclude the discussion on trajectory tracking via DFL by offering some remarks :

— The state of the dynamic compensator should be correctly initialized at the value $\xi(0) = v_d(0)$. This guarantees exact trajectory tracking for a matched initial state of the robot. In this case, the control law (1.49), (1.52) reduces to the pure feedforward action.

— Being based purely on an output tracking error definition, this method requires neither the explicit computation of $\theta_d(t)$ nor the measure of the orientation angle $\theta(t)$.

— Even for smooth persistent trajectories, problems may arise if the actual command $v = \xi$ crosses zero during an initial transient. However, this situation can be avoided by suitably choosing the initial state of the dynamic compensator. For example, a simple way to keep the actual commands bounded is to reset the state ξ whenever its value falls below a given threshold. This strategy results in an input command v with isolated discontinuities with respect to time. [7]

1.6 Conclusion

As we have seen in this chapter. First, we presented a WMRs kinematic model under the non-holonomic constraint. Next, path planning for non-holonomic mobile robots was discussed. After defining a flat output system and the features incorporated with it, trajectory planning via the Cartesian polynomial method was fully explained. Finally, linear, nonlinear and DFL controls were presented.

The implementation of the latter control strategies and the planning method for the real robot is difficult. But there's a way to simplify the process. The next chapter will introduce the Robot Operating System ROS.

CHAPTER 2

Robot Operating System

2.1 Introduction

Robots do still present some significant challenges for software developers. This chapter introduces a software platform called Robot Operating System, or ROS, that is intended to ease some of these difficulties. Although ROS is not an operating system (OS) but a set of software frameworks for robot software development. The official description of ROS is:

ROS is an open-source, meta-operating system for robots. It provides the services we would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS is similar in some respects to 'robot frameworks,' such as Player, YARP, Orocos, CARMEN, Orca, MOOS, and Microsoft Robotics Studio. ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems, though the ROS community has been contributing support for Fedora, Gentoo, Arch Linux and other Linux platforms. While a port to Microsoft Windows for ROS is possible, it has not yet been fully explored. [25]. In this chapter, we present briefly the history of ROS, the main philosophy behind its mode of functioning, simulators based on ROS, and many other functionalities.

Brief History

ROS is a large project with many ancestors and contributors. Many in the robotics research community felt the need for an open collaboration framework. The first pieces of what eventually would become ROS began coalescing at Stanford University. Eric Berger and Keenan Wyrobek, PhD students working in Kenneth Salisbury's robotics laboratory at Stanford, were leading the Personal Robotics Program. While working on robots to do manipulation tasks in human environments, the two students noticed that many of their colleagues were held back by the diverse nature of robotics.

In 2007, Eric and Keenan joined Willow Garage, Inc., a nearby robotics incubator, and provided significant resources to extend these concepts much further and create well-tested implementations. The effort was boosted by countless researchers who contributed their time

and expertise to the core of ROS and its fundamental software packages. Throughout, the software was developed in the open using the permissive BSD open source license, and it gradually became widely used in the robotics research community.

From the start, ROS was being developed at multiple institutions and for multiple robots. At first, this seemed like a headache, since it would have been far simpler for all contributors to place their code on the same servers. Ironically, over the years, this has emerged as one of the great strengths of the ROS ecosystem: any group can start their own ROS code repository on their own servers, and they will maintain full ownership and control of it. They don't need anyone's permission. If they choose to make their repository publicly visible, they can receive the recognition and credit they deserve for their achievements and benefit from specific technical feedback and improvements like all open source software projects. [20] Since 2013 managed by OSRF (Open Source Robotics Foundation).

The ROS ecosystem is now very popular among roboticists. Researchers, hobbyists, and even robotics companies are using it, promoting it and supporting it.

Philosophy

ROS follows the Unix philosophy of software development in several important aspects. This makes ROS feel "natural" to developers with a Unix background, but a bit "mysterious" at first to the developers using the Windows or Mac OSX graphical development environment. The next paragraphs describe some of the philosophical aspects of ROS:

Peer to peer ROS systems consist of numerous small computer programs that connect to one another and continuously exchange messages. These messages travel directly from one program to another; there is no central routing service. Although this makes the underlying "plumbing" more complex, the result is a system that scales better as the amount of data increases. [20]

Tools-based In an effort to manage the complexity of ROS, we have opted for a microkernel design, where a large number of small tools are used to build and run the various ROS components, rather than constructing a monolithic development and runtime environment.

These tools perform various tasks, e.g., navigate the source code tree, get and set configuration parameters, visualize the peer-to-peer connection topology, measure bandwidth utilization, graphically plot message data, auto-generate documentation, and so on. Although we could have implemented core services such as a global clock and a logger inside the master

module, we have attempted to push everything into separate modules. We believe the loss in efficiency is more than offset by the gains in stability and complexity management. [19]

Multi-lingual Many software tasks are easier to accomplish in "high-productivity" scripting languages such as Python or Ruby. However, there are times when performance requirements dictate the use of faster languages, such as C++. There are also various reasons that some programmers prefer languages such as Lisp or MATLAB. Endless email flame wars have been waged, are currently being waged, and will doubtless continue to be waged over which language is best suited for a particular task. Acknowledging that all of these opinions have merit, that languages have different utilities in different contexts, and that each programmer's unique background is hugely important when choosing a language, ROS chose a multilingual approach. ROS software modules can be written in any language for which a client library has been written. At the time of writing, client libraries exist for C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia, and others. ROS client libraries communicate with one another by following a convention that describes how messages are "flattened" or "serialized" before being transmitted over the network. This book will use the Python client library almost exclusively, to save space in the code examples and for its general ease of use. However, the tasks described in this book can be accomplished with any of the client libraries. [20]

Thin The ROS conventions encourage contributors to create standalone libraries and then wrap those libraries so they can send and receive messages to and from other ROS modules. This extra layer is intended to allow the reuse of software outside of ROS for other applications, and it greatly simplifies the creation of automated tests using standard continuous integration tools. [20]

Free and open source The core of ROS is released under the permissive BSD license, which allows commercial and noncommercial use. ROS passes data between modules using interprocess communication (IPC), which means that systems built using ROS can have fine-grained licensing of their various components. Commercial systems, for example, often have several closed source modules communicating with a large number of open source modules. Academic and hobby projects are often fully open source. Commercial product development is often done completely behind a firewall. All of these use cases, and more, are common and perfectly valid under the ROS license. [20]

2.2 The fundamental concepts of ROS

ROS is divided into three conceptual levels: the filesystem level, the computation graph level, and the community level.

2.2.1 Filesystem Level

As mentioned in [13]. ROS is more than a development framework. We can refer to ROS as a meta-operating system, since it offers not only tools and libraries but even OS-like functions, such as hardware abstraction, package management, and a developer toolchain. Like a real operating system, ROS files are organized on the hard disk in a particular manner, as depicted in the following figure:

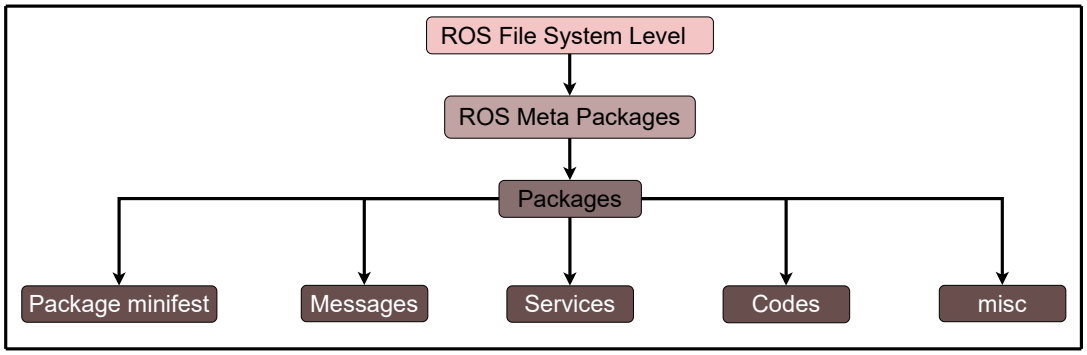

Figure 2.1 ROS filesystem level. [13]

Here are the explanations for each block in the filesystem:

Packages: The ROS packages are the most basic unit of the ROS software. They contain one or more ROS programs (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software. [13] A typical structure of an ROS package is shown here:

Where, the $< config >$ folder is created by the user to store all the configuration files that are used in this ROS package, the *include* folder consists of headers and libraries that we need to use inside the package, *script* keeps executable Python scripts, *src* stores the C++ source codes, the *launch* folder keeps the launch files that are used to launch one or more ROS nodes, and *msg* contains custom message definition. The *srv* folder contains the services definition, the *action* folder contains the action files. Finally, the $CMakeLists.txt$ files contains the directives to compile the package. [13]

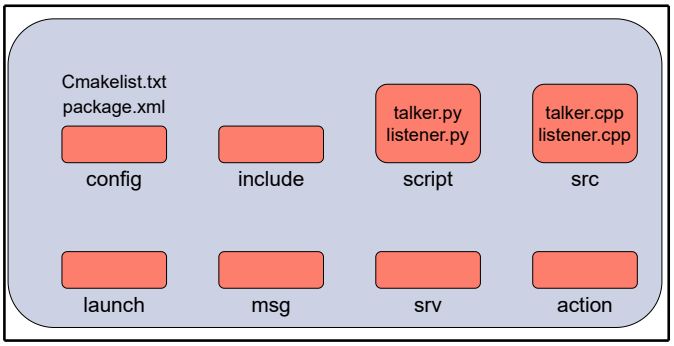

Figure 2.2 Structure of a typical ROS package. [13]

Package manifest: The package manifest file is inside a package that contains information about the package, author, license, dependencies, compilation flags, and so on. The package.xml file inside the ROS package is the manifest file of that package. [13]

Metapackages: The term metapackage refers to one or more related packages which can be loosely grouped together. In principle, metapackages are virtual packages that don't contain any source code or typical files usually found in packages. And the metapackage manifest is similar to the package manifest, the difference being that it might include packages inside it as runtime dependencies and declare an export tag. [13]

Messages (*.msg*): The ROS messages are a type of information that is sent from one ROS process to the other. We can define a custom message inside the msg folder inside a package ($my\_package/msg/MyMessageType.msg$). The extension of the message file is *.msg*. [13]

Services(*.srv*) The ROS service is a kind of request/reply interaction between processes. The reply and request data types can be defined inside the srv folder inside the package ($my\_package/srv/MyServiceType.srv$). [13]

Repositories Most of the ROS packages are maintained using a Version Control System (VCS), such as Git, Subversion (svn), Mercurial (hg), and so on. The collection of packages that share a common VCS can be called repositories. The package in the repositories can be released using a catkin release automation tool called bloom. [13]

The following figure gives an idea of the files and folders of a package inside the ROS universe:

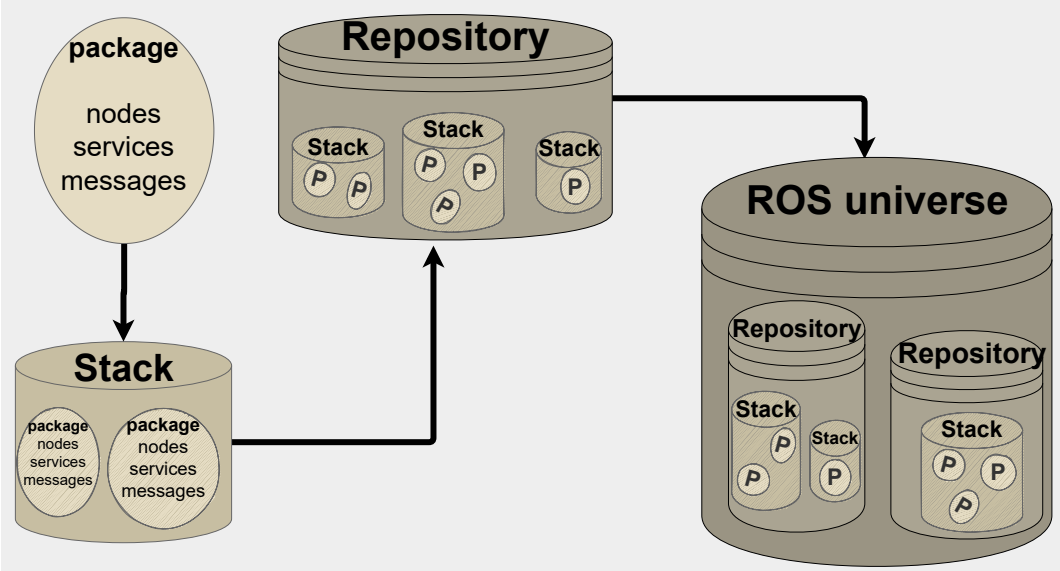

Figure 2.3 ROS package system.

2.2.2 Computation Graph Level

According to [14]. The ROS computation graph is the peer-to-peer network of the ROS process, and it processes the data together. The ROS computation graph concepts are nodes, topics, messages, master, parameter server, services, and bags:

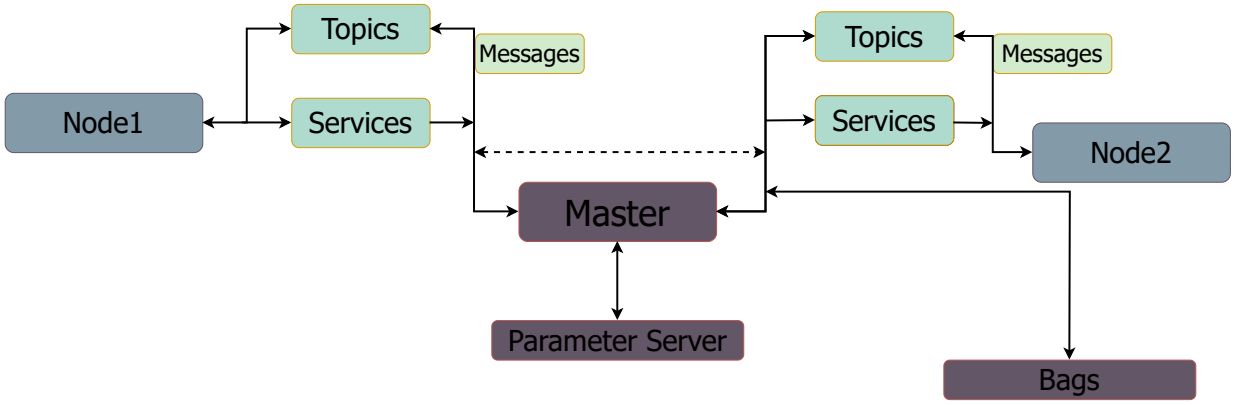

Figure 2.4 The ROS computational graph concept diagram. [14]

The preceding figure (2.4) shows the various concepts in the ROS computational graph. Here is a short description of each concept:

Nodes: ROS nodes are simply a process that is using ROS APIs to communicate with each other. A robot may have many nodes to perform its computations. For example, an autonomous mobile robot may have a node each for hardware interfacing, reading laser scans, and localization and mapping. We can create ROS nodes using ROS client libraries such as *roscpp* and *rospy*. [14]

Master: The ROS master works as an intermediate node that aids connections between different ROS nodes. The master has all the details about all nodes running in the ROS environment. It will exchange details of one node with another in order to establish a connection between them. After exchanging the information, communication will start between the two ROS nodes. [14]

Parameter server: The parameter server is a pretty useful thing in ROS. A node can store a variable in the parameter server and set its privacy too. If the parameter has a global scope, it can be accessed by all other nodes. The ROS parameter runs along with the ROS master. [14]

Messages: ROS nodes can communicate with each other in many ways. In all methods, nodes send and receive data in the form of ROS messages. The ROS message is a data structure used by ROS nodes to exchange data. [14]

Topics: One of the methods to communicate and exchange ROS messages between two ROS nodes is called ROS topics. Topics are named buses, in which data is exchanged using ROS messages. Each topic will have a specific name, and one node will publish data to a topic and an other node can read from the topic by subscribing to it. [14]

Services: Services are another kind of communication method, like topics. Topics use publish or subscribe interaction, but in services, a request or reply method is used. One node will act as the service provider, which has a service routine running, and a client node requests a service from the server. The server will execute the service routine and send the result to the client. The client node should wait until the server responds with the results. [14]

Bags: Bags are a useful utility in ROS for the recording and playback of ROS topics. While working on robots, there may be some situations where we need to work without actual hardware. Using *rosbag*, we can record sensor data and can copy the bag file to other computers to inspect data by playing it back. [14]

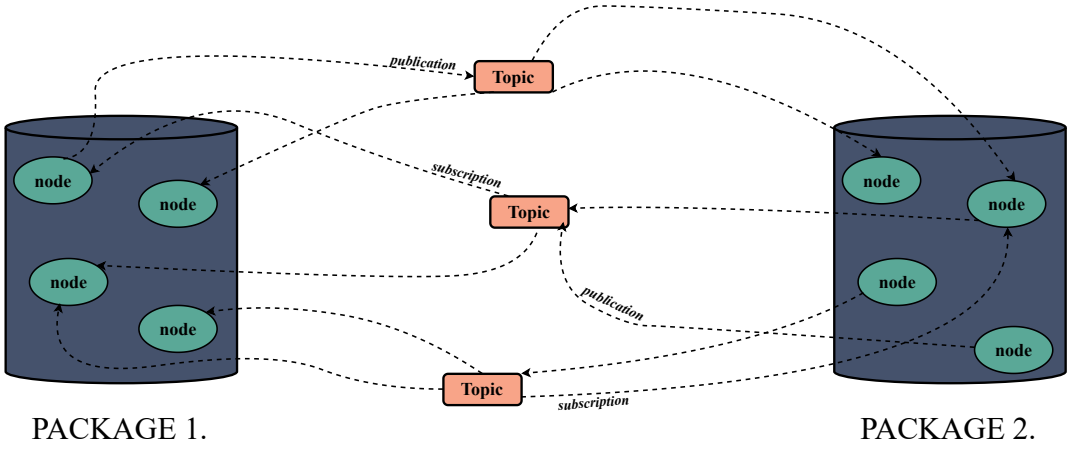

Figure 2.5 Model of how the ROS nodes publish and subscribe to topics.

2.2.3 Community Level

The ROS Community Level consists of ROS distributions, repositories, the ROS Wiki, and ROS Answers, which enable researchers, hobbyists, and industries to exchange software, ideas, and knowledge in order to progress robotics communities worldwide. ROS distributions are similar to the roles that Linux distributions play. They are a collection of versioned ROS stacks, which allow users to utilize different versions of ROS software frameworks. Even while ROS continues to be updated, users can maintain their projects with older more stable versions and can easily switch between versions at any time.

ROS does not maintain a single repository for ROS packages; rather, ROS encourages users and developers to host their own repositories for packages that they have used or created. ROS simply provides an index of packages, allowing developers to maintain ownership and control over their software. Developers can then utilize the ROS Wiki to advertise and create tutorials to demonstrate the use and functionality of their packages. The ROS Wiki is the forum for documenting information about ROS, where researchers and developers contribute documentation, updates, links to their repositories, and tutorials for any open-sourced software they have produced. ROS Answers is a community-oriented site to help

answer ROS-related questions that users may have.

The following figure shows some of the latest ROS distributions:

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017

Figure 2.6 ROS distributions.

2.2.4 Other ROS Concepts

2.2.4.1 URDF

The unified robot description format (URDF) package contains an XML file that represents a robot model. The URDF is another tool within ROS that makes it a modular system. Rather than creating a unique process for different styles of robots, nodes are created without regard for the robot that will utilize them. The URDF file provides the necessary, robot-specific, information so nodes may conduct their procedures. A URDF file is written so that each link of the robot is the child of a parent link, with joints connecting each link, and joints are defined with their offset from the reference frame of the parent link and their axis of

rotation [2]. In this way, a complete kinematic model of the robot is created. A tree diagram can be visualized utilizing the *urdf_to_graphiz* tool as is shown in fig((2.7)).

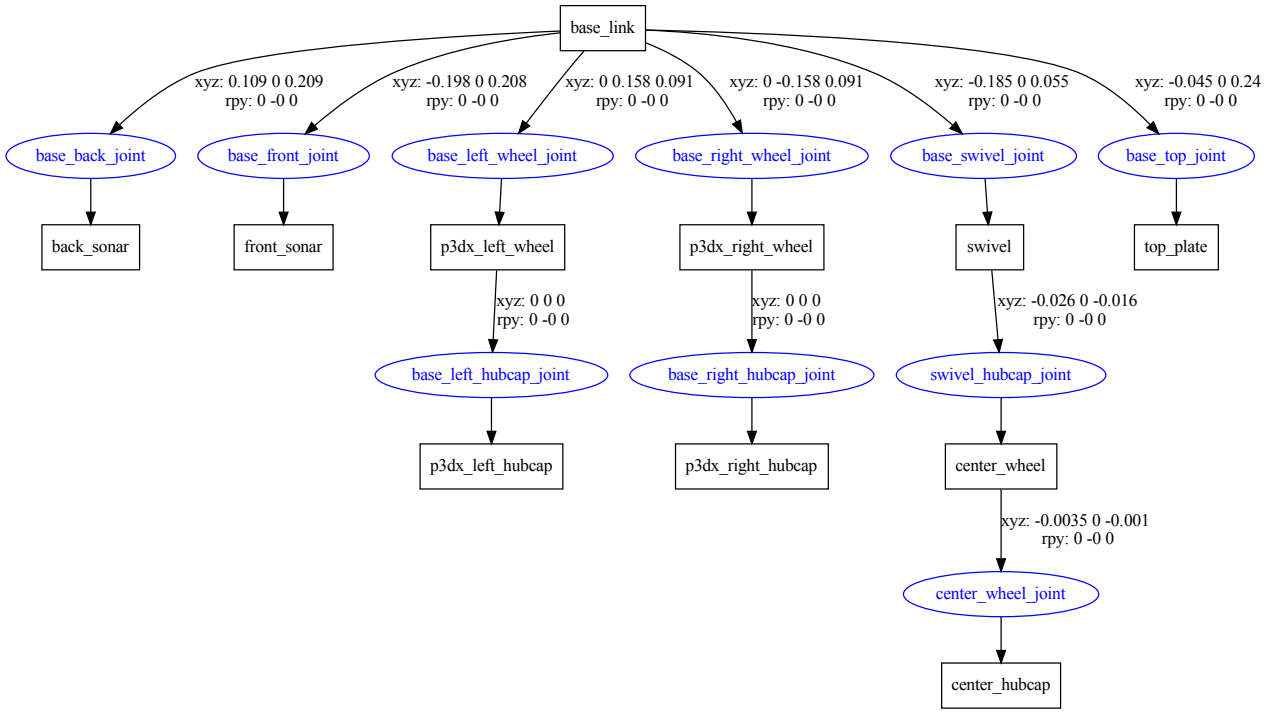

Figure 2.7 A diagram of URDF of the P3-DX is shown utilizing the urdf-to-graphiz tool.

2.2.4.2 Coordinate Frames and Transforms.

A robotic system typically has many three-dimensional coordinate frames that change over time. The tf ROS package keeps track of multiple coordinate frames in the form of a tree structure. Just as the URDF manages joints and links, the tf package maintains the relationships between coordinate frames of points, vectors, and poses, and computes the transforms between them. The tf package operates in a distributed system, all ROS components within the system have access to information about the coordinate frames. The transform tree can also be viewed by developers for debugging by utilizing the $view\_frames$ tool as shown in Fig((2.8)). Additional command-line tools for the tf package are $rosrun\ tf\ tf\_monitor$, $rosrun\ tf\ tf\_echo < source\_frame >< target\_frame >$, and $roswtf$, which, respectively, monitors delays between transforms of coordinate frames, prints transforms between coordinate frames, and aids in debugging [5].

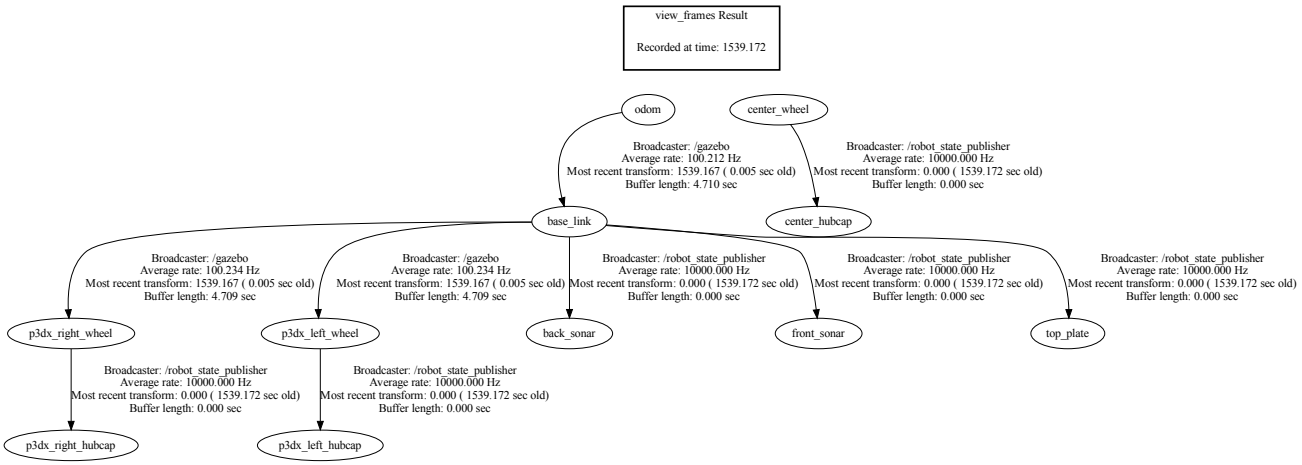

Figure 2.8 The transforms tree diagram of the P3-DX is shown utilizing the view-frames tool.

2.2.5 Basic ROS Commands

ROS provides users with a variety of tools in order to make navigation through the ROS filesystem and debugging as simple as possible. A few basic ROS commands utilized within this thesis are shown in Table 1.

roscore	Starts ROS Master.
rosrun $< pkg\_name > < node\_name >$	Starts executable node.
roslaunch $< pkg\_name > < launch\_file >$	Starts launch file.
rostopic *list*	Lists all active topics.
rostopic info $< /topic\_name >$	Provides data on topic such as type, subscribers and publishers.
rostopic echo $< /topic\_name >$	Prints topic messages to screen.
rostopic hz $< /topic\_name >$	Prints publishing rate to screen.
rosnode *list*	Lists all nodes running.
rosnode info $< node\_name >$	Provides data on node such as publications, subcriptions, services, and Pid.
Rosmsg show –r $< msg\_type >$	Prints raw message text.
rospack find $< package\_name >$	Prints file path to package.
rosrun *rqt_graph rqt_graph*	Tool to visualize graphical representation of active packages, nodes, and topics.

Rosbag record –O <filename> </topic>	Starts rosbag tool to record data from a desired topic.

Table 2.1 – Basic ROS Commands

2.3 Gazebo Simulator

One of the open source robotic simulators tightly integrated with ROS is Gazebo (`http://gazebosim.org`). Gazebo is a dynamic robotic simulator that has a wide variety of robot models and extensive sensor support. The functionalities of Gazebo can be added via plugins. The sensor values can be accessed by ROS through topics, parameters, and services. Gazebo can be used when our simulation needs full compatibility with ROS. Most of the robotics simulators are proprietary and expensive, since we can't afford it, we can use Gazebo directly without any issues. [9]

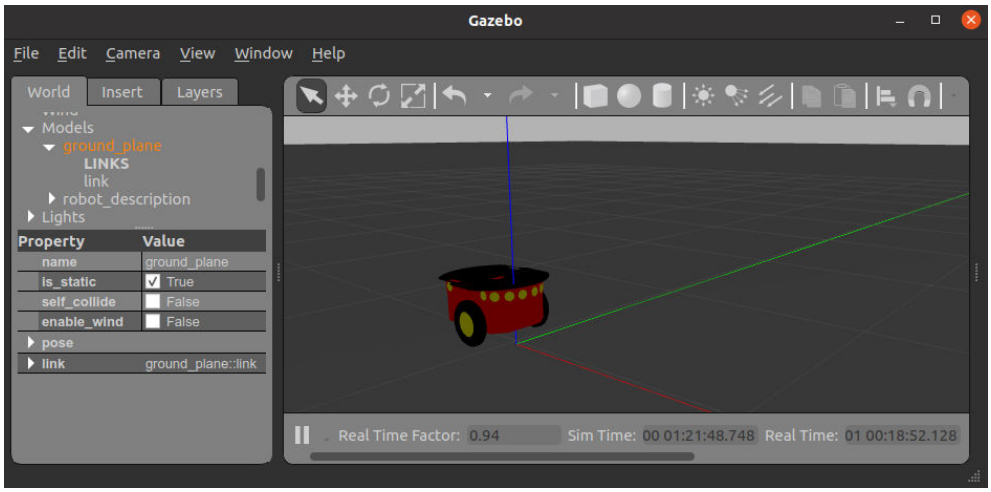

Figure 2.9 P3DX robot in Gazebo Simulator.

2.4 Robot hardware

The Pioneer 3-DX mobile robot used in this work is presented in figure ((2.10)). It is a wheeled differential drive mobile robot employed to evaluate the proposed control method.

Figure 2.10 P3-DX mobile robot.

Pioneer robots may be smaller than most, but they pack an impressive array of intelligent mobile robot capabilities that rival bigger and much more expensive machines. The P3-DX with onboard PC is a fully autonomous intelligent mobile robot. Pioneer's modest size lends itself very well to navigation in tight quarters and cluttered spaces, such as classrooms, laboratories, and small offices. The P3-DX is fully capable of mapping its environment, finding its way home and performing other sophisticated path-planning tasks.

Weighing only 9 kg (with one battery), the basic P3-DX mobile robot is lightweight, but his strong aluminum body and solid construction makes it virtually indestructible. These characteristics also permit it to carry extraordinary payloads, The P3-DX can carry up to 23 Kg additional weight [21], the physical dimensions are shown in figure ((2.11)).

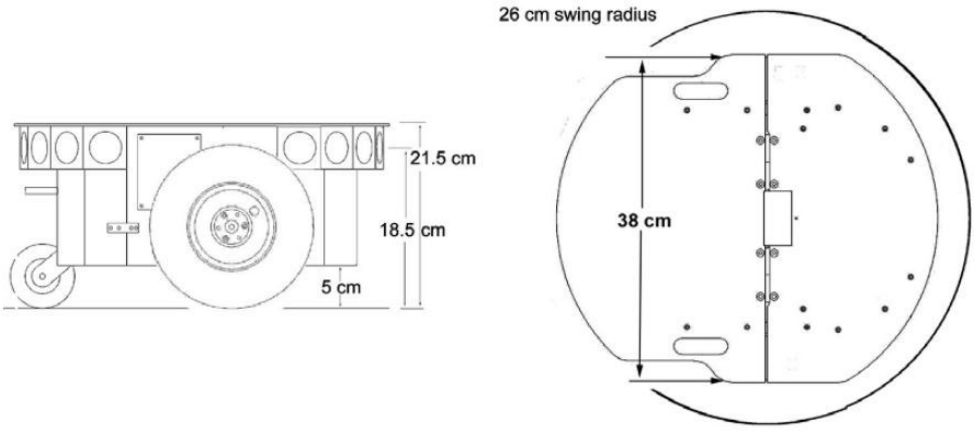

Figure 2.11 Physical dimensions of the robot. [21]

2.5 Robot drivers

In order to communicate with and control the P3-DX using ROS, it is necessary to install the appropriate drivers. There are two main drivers available for the Adept MobileRobots Pioneer family of robots, the $p2os$ and $ROSARIA$ stacks.

2.5.1 RosAria stack

The $RosAria$ node authored by $Srećko Jurić - Kavelj$ is available for ROS distributions. The node provides an interface with ROS for Adept MobileRobots which use the open source $ARIA$ library. First, the $ARIA$ library must be installed from Adept MobileRobots or its alternative version $AriaCoda$. [11] The $ARIACoda$ software for Ubuntu 20.04.4, for a 64- bit architecture was downloaded. The $RosAria$ node was cloned from source using git. By using rosdep, we also installed the necessary dependencies. Lastly, the node was built using catkin. To test and run the $RosAria$ node, the "How to use ROSARIA" tutorial is referenced. [10]

The $RosAria$ node publishes on multiple topics including pose and battery voltage. In this section we only consider topics relevant to the control objective. The $/pose$ topic is of the ROS message type $nav\_msgs/Odometry$. The message contains the position in xyz-coordinates as well as the rotation of the robot. Both the pose and twist messages also contain covariance values. The pose of the robot is set to $[0, 0, 0]$ at the start of the $RosAria$ node. The battery voltage is a float message of type $std\_msgs/Float32$ that gives a measurement of the battery voltage in DC. The node subscribes to the topic $/cmd\_vel$, the topic on which new velocity commands to the robot can be published. When a command is sent over the topic, the desired velocity is set in ARIA. The robot obtains and maintains the velocity sent for up to 600 ms unless another velocity command is received. As such, the velocity commands to the robot are only necessary for changing its speed. [11]

With the mobile robot connected to the computer using a RS232 serial communications link, figure (2.12) clarify the necessary Terminal Window Command Lines to build and run the $RosAria$ node.

```
# Bring ROSARIA into the workspace
$ cd ~/catkin_ws/src
$ git clone https://github.com/amor-ros-pkg/rosaria.git
$ cd ..
$ catkin_make

# Running RosAria
# Tab 1 (roscore)

$ export ROS_MASTER_URI=http://192.168.1.1:11311
$ export ROS_IP=192.168.1.1
$ sudo chmod 777 -R /dev/ttyUSB0     #or (/dev/ttyS0)
$ roscore
# Tab 2 (rosaria node)
$ source catkin_ws/devel/setup.sh
$ rosrun rosaria RosAria
```

Figure 2.12 Terminal Window Command Lines to build and run the *RosAria* node. [10]

2.5.2 P2os stack

This stack is considered as alternative to the Rosaria package, and was designed to replace Rosaria and provide other utilities such as simulation model in Gazebo and tele-operation node. The packages that comprise the *p2os* stack are *p2os_driver*, *p2os_launch*, *p2os_teleop*, *p2os_urdf*, and *p2os_msgs*.

The *p2os_driver* package is the main package of the *p2os* stack and contains nodes, libraries, and parameters that are essential for ROS to interface with the P3-DX's client-server Advanced Robot Control and Operations Software (ARCOS). The package *p2os_driver* receives linear and angular velocity commands by subscribing to the ROS topic */cmd_vel* and sends the necessary motor commands to the P3-DX. Additionally, the *p2os_driver* package extracts motor encoder information and publishes position, orientation, and velocity in the form of an odometry message from to the ROS topic */pose*. [15]

The *p2os_driver* package is also responsible for publishing the transforms of the robot to the /tf topic for other ROS nodes to utilize. Additionally, the *p2os_driver* package publishes useful information about the P3-DX such as its battery state, digital input/output voltage, and analog input/output voltage . The *p2os_driver* package utilizes the URDF from the *p2os_urdf* package. The URDF is responsible for establishing the transforms for each joint and link of the robot so the *p2os_driver* package can publish it to the /tf topic. The *p2os_driver* package also utilizes message formats that are specific to the *p2os* stack, which

are located in the *p2os_msgs* package.

The *p2os_launch* package contains useful ROS launch files for running multiple nodes of the p2os stack to systematically start certain operations of the robot. Some of the launch files run necessary parameters for proper navigation of the P3-DX, while others are used for running sensors such as the Hokuyo laser range scanner. This package also provides a full launch file for launching P3-DX in Gazebo simulator. The model in this simulator can subscribe to and/or publish messages in the same topics as the real mobile robot.

```
# Bring P2OS into the workspace
$ cd ~/catkin_ws/src
$ git clone https://github.com/allenh1/p2os.git
$ cd ..
$ catkin_make

# Running P3DX.gazebo.launch
$ source catkin_ws/devel/setup.sh
$ roslaunch p2os_urdf pioneer3dx.gazebo.launch
```

Figure 2.13 Command Lines to build the *P2OS* package and run the P3-DX on gazebo. [15]

Note 1 *In this work, the RosAria package is used as the main driver to control the robot P3DX. Whereas, the P2os package is used only for simulation of the model in Gazebo simulator. It is worth mentioning that the P2os package can be also as driver to control the robot, however, due to technical problems the was not possible to validate in this work.*

2.5.3 Teleop keyboard package

The package *teleop_twist_keyboard* authored by Graylin Trevor Jay offers the option of controlling the robot via the keyboard. [18] Figure (2.14) illustrate the essential Command Lines to install and run this package.

```
# Installing
$ sudo apt-get install ros-noetic-teleop-twist-keyboard

#Running
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/
   RosAria/cmd_vel

#Controls
  Reading from the keyboard  and Publishing to Twist!
  ---------------------------
  Moving around:
   u    i    o
   j    k    l
   m    ,    .

   q/z : increase/decrease max speeds by 10%
   w/x : increase/decrease only linear speed by 10%
   e/c : increase/decrease only angular speed by 10%
  anything else : stop
  CTRL-C to quit
```

Figure 2.14 Command Lines to install and run *teleop_twist_keyboard*. [18]

2.6 Odometry

The implementation of any feedback controller requires the availability of the robot configuration at each time instant. Mobile robots are equipped with incremental encoders that measure the rotation of the wheels, this information along with robot model can be used to estimate the robot position and orientation. To this end, the following equations are used to provide a good estimates for robot pose

$$\begin{aligned} x_{k+1} &= x_k + \frac{v_k}{\omega_k}\left(\sin\theta_{k+1} - \sin\theta_k\right) \\ y_{k+1} &= y_k - \frac{v_k}{\omega_k}\left(\cos\theta_{k+1} - \cos\theta_k\right) \\ \theta_{k+1} &= \theta_k + \omega_k T_s \end{aligned} \tag{2.1}$$

where T_s is the sampling time. A full disscussion on the odometry analysis can be found in [23]

Both the Rosaria node and the P2OS stack extract motor encoder information and publishes position, orientation, and velocity in the form of an odometry message, Figure (2.15) shows the expanded Definition of the *nav_msgs/Odometry* Message

```
geometry_msgs/Pose pose
geometry_msgs/Point position
float64 x
float64 y
float64 z
geometry_msgs/Quaternion orientation
float64 x
float64 y
float64 z
float64 w
float64[36] covariance
```

Figure 2.15 *nav_msgs/Odometry* Message. [1]

AS we can see. The orientation is in quaternion format, a quaternion is one of several mathematical ways to represent the orientation and rotation of an object in three dimensions. Quaternions are often used instead of Euler angle rotation matrices because compared to rotation matrices they are more compact, more numerically stable, and more efficient. From the Figure (2.15), the quaternion variables in *geometry_−msgs/Quaternion orientation* provides the component of the rotation axis unit vector x, y, z, and the w is the variable

related to the angle of rotation

$$\begin{aligned} \mathrm{w} &= \frac{cos(\theta)}{2} \qquad \text{Which yeilds} \\ \theta &= 2acos(\mathrm{w}) \end{aligned} \tag{2.2}$$

On the other hand the variables in *geometry_msgs/Point position* provides the component of the position vector of the robot with respect to the world frame, the relevent components are the x, y.

2.7 Conclusion

In this chapter, An explanation of ROS, and the P3-DX hardware are provided. The integration of ROS software with the P3-DX, as well as the odometry approaches are discussed.

The focus of next Chapter is the results, and the effectiveness of the integration of the control laws presented in the previous chapter, and the ROS packages to the P3-DX.

CHAPTER 3

Simulation and Experiment resultsts

3.1 Introduction

Having described ROS in the previous chapter, we now proceed to implement the trajectory planning method and the motion control strategies explained in the first chapter into the P3-DX robot using the *p2os* stack for simulation on *gazebo*, and the *RosAria* stack for the real robot.

Trajectory planning and motion control are simulated in MATLAB/SIMULINK (R2020b), the kinematic model (1.7) was used to generate the desired trajectory, where its inputs (v_d and ω_d) were computed as demonstrated in (section.1.4.4). Thanks to the ROS toolbox in SIMULINK, the pose of the robot was received by subscribing to the */pioneer/odom* topic in simulation and the */RosAria/pose* topic in experiment with regard to the error $(q_d - q)$ which is then fed to the controllers (section 1.5.1, section 1.5.2, and section 1.5.3.). Sending the appropriate commands can be done by publishing in the */pioneer/cmd_vel* topic in simulation, and */RosAria/cmd_vel* in experiment.
The non-linear simulation scheme for example is illustrated in a block diagram as shown in Figure 3.1

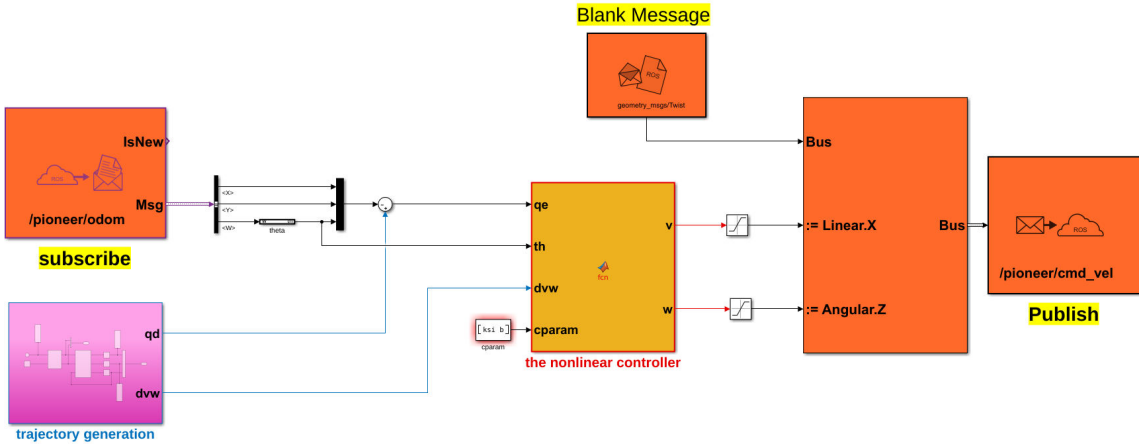

Figure 3.1 Simulink block diagram of simulation scheme.

3.2 Simulations results

Before implementing the controllers on the real robot, a simulation was conducted first to assess the effectiveness of the trajectory tracking controllers. The *P2OS* stack and the environment *GAZEBO* were used for simulating a virtual "P3-DX", seeking to track a desired path produced by the planner that uses cubic Cartesian polynomials, where the free parameter k in (1.25) was chosen equal to 5, $t \in [0, 20]$, and the simulation sampling time equal to 30 ms. The initial and final configurations are illustrated in the following table:

	x	y	θ
initial pose	0.2	0.2	0
final pose	4	1.5	$\pi/3$

Table 3.1 The desired path configurations.

3.2.1 Linear controller simulation results

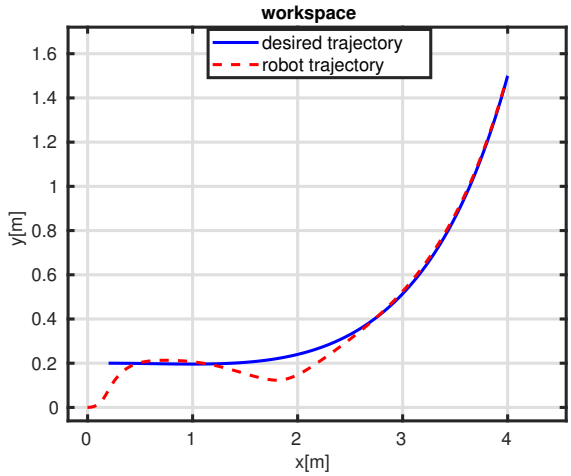

Figure 3.2 Linear control simulation results : Desired and robot trajectories.

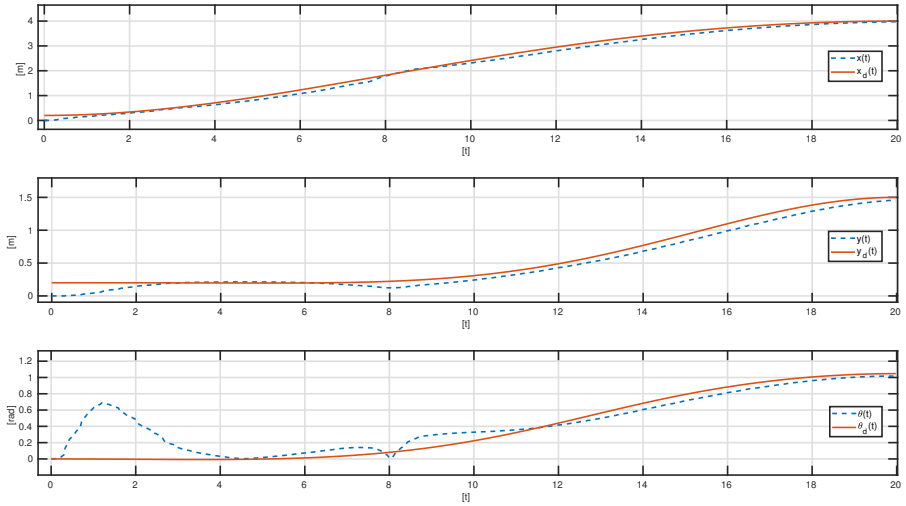

Figure 3.3 Linear control simulation results : Desired and robot states.

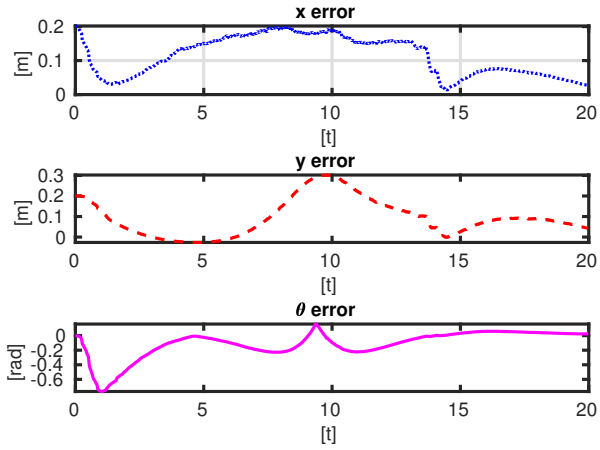

Figure 3.4 Linear control simulation results : State errors

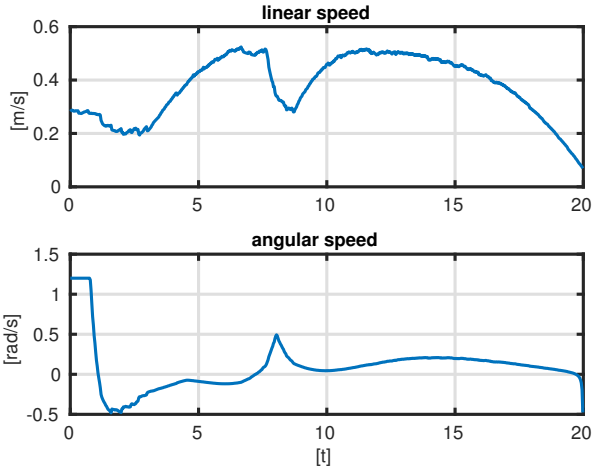

Figure 3.5 Linear control simulation results : Velocities

3.2.2 NonLinear controller simulation results

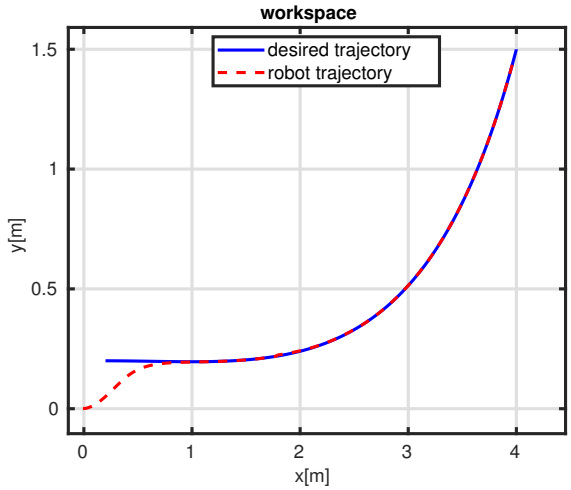

Figure 3.6 Nonlinear control simulation results : Desired and robot trajectories.

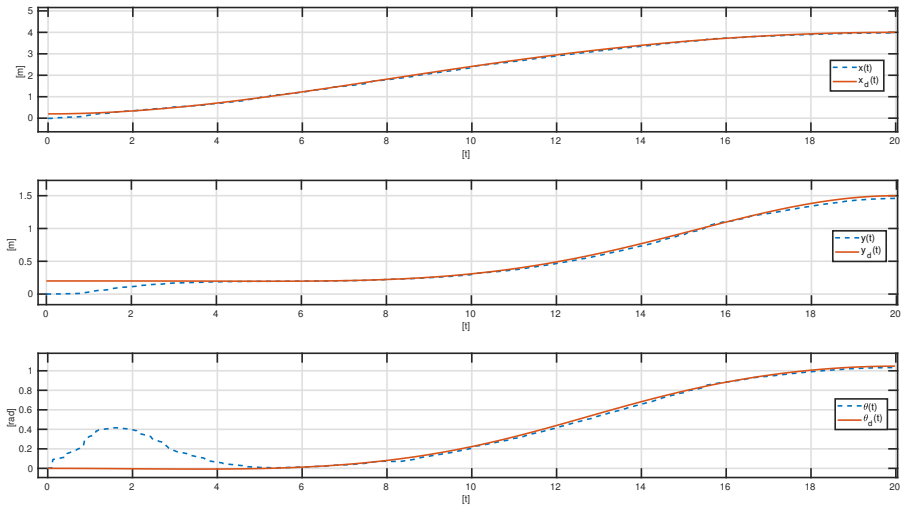

Figure 3.7 Nonlinear control simulation results : Desired and robot states.

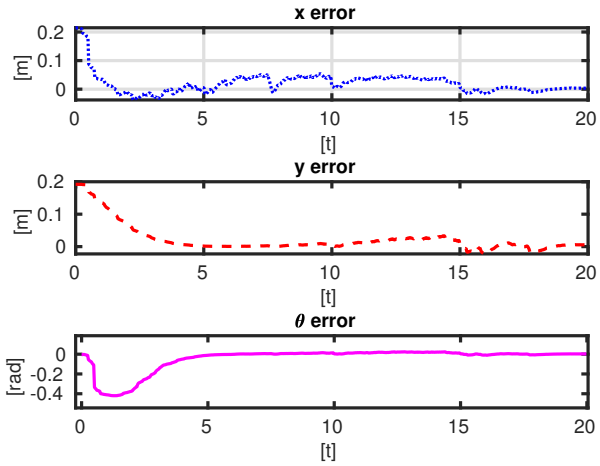

Figure 3.8 Nonlinear control simulation results : State errors

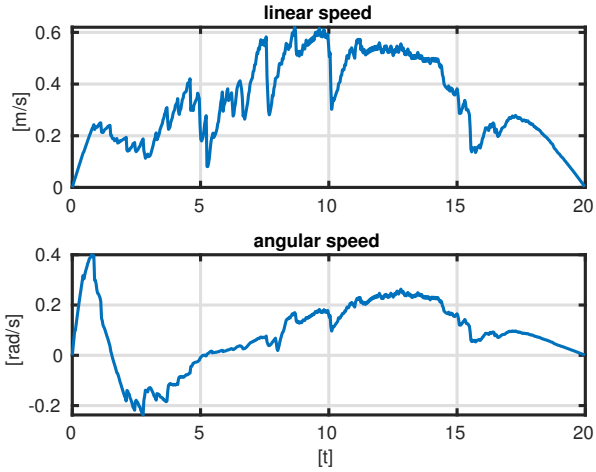

Figure 3.9 Nonlinear control simulation results : Velocities

3.2.3 Dynamic feedback linearization controller simulation results

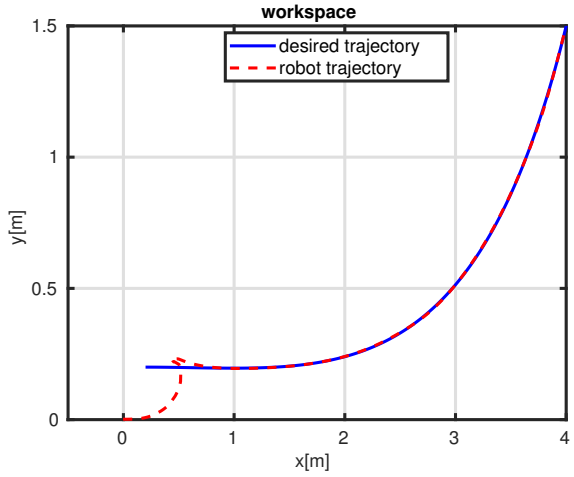

Figure 3.10 DFL simulation results : Desired and robot trajectories.

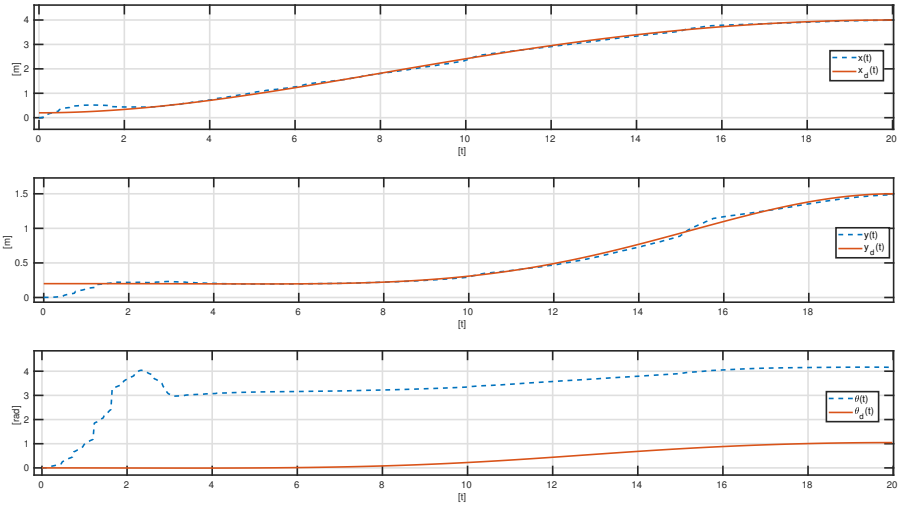

Figure 3.11 DFL simulation results : Desired and robot states.

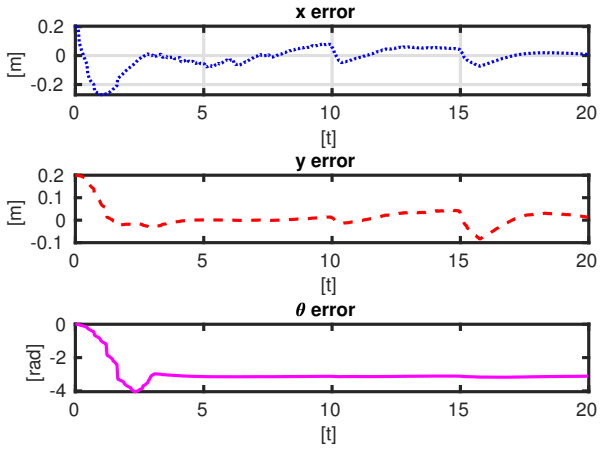

Figure 3.12 DFL simulation results : State errors

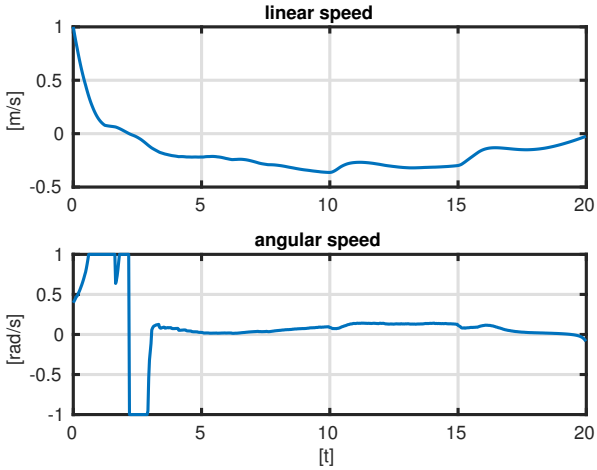

Figure 3.13 DFL simulation results : Velocities

3.3 Results discussion

Figures 3.2, 3.3, 3.4, and 3.5 show the simulation results obtained with the linearly designed controller 1.32 , using the gains 1.35 with $\zeta = 0.7$ and a= 1. The tracking of the reference trajectory as shown in figure 3.2 is not quite accurate, where as demonstrated figure 3.3 in the 8^{th} second, the robot trajectory is slightly distracted from the desired trajectory. Residual errors in the figure 3.4 are mainly due to the nonidealities. In particular, the large

transient error because of the initial non-zero value of v(0) as shown in figure 3.5.

Better performance is obtained with the nonlinear controller 1.38, using the parameters for the gains 1.42 with ζ=1 and b=70. As pointed out in figure 3.6 the tracking is quite precise. The tracking errors in figure 3.8 converges to some negligible values around zero.

Although we have obtained some excellent results with the DFL control law using the gains (1.52) $k_{p1} = k_{p2} = k_{d1} = k_{d2} = 2$, and while canceling the error of the two states x and y, the θ error is equal to $-\pi$ (Tracking with backward motion) as we can see in Figure 3.12, the explanation behind this is that the design of the DFL require more attention with regard to the design parameters because of the problem of singularities discussed in chapter 1. To solve this problem the DFL should contain more synthesis constraints which are not presented here.

3.4 Experimental results

After validating the control performance in simulation with gazebo model, we show now the real performance of the proposed controllers on the real platform P3-dx. The same control strategies with the same design parameters were used on the mobile platform with minor changes including the sampling time which was modified to 100ms and the duration of the mobile platform is set to 100s. However, the Rosaria package was used for driving the robot instead of P2os package.

For the linear control in Figures 3.14,3.15,3.16, and 3.17, the used gains are $\zeta = 0.5$, b=0.2, as for the non-linear controller in Figures 3.18,3.19,3.20, and3.21 $\zeta = 1$, b=70, finally, for the DFL method in Figures 3.22,3.23,3.24, and 3.25, the gains are as follows : $k_{p1} = k_{p2} = k_{d1} = k_{d2} = 5$.

3.4.1 Linear control experimental results

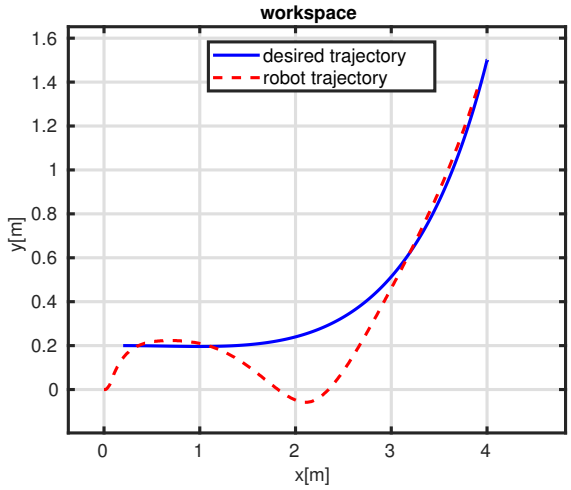

Figure 3.14 Linear control experimental results : Desired and robot trajectories.

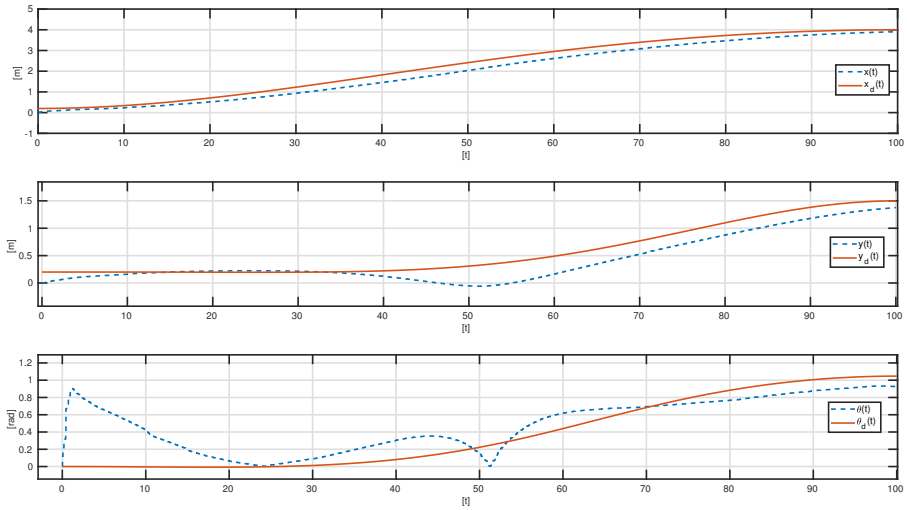

Figure 3.15 Linear control experimental results : Desired and robot states.

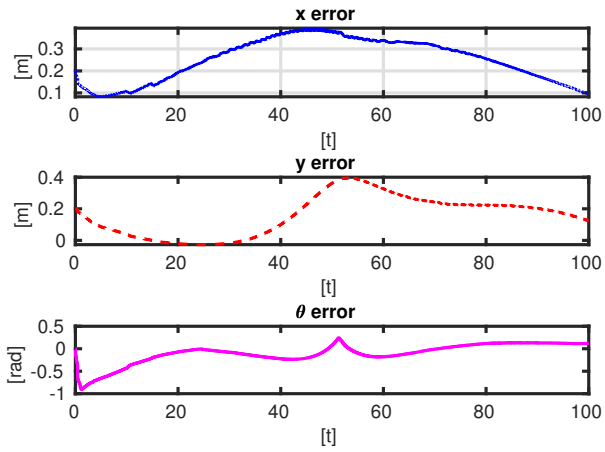

Figure 3.16 Linear control experimental results : State errors

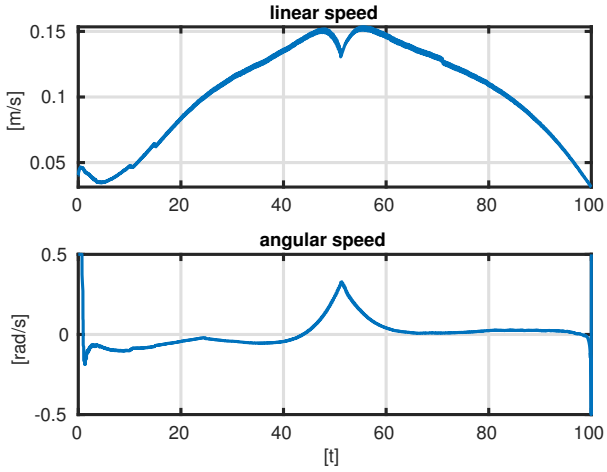

Figure 3.17 Linear control experimental results : Velocities

3.4.2 Nonlinear control experimental results

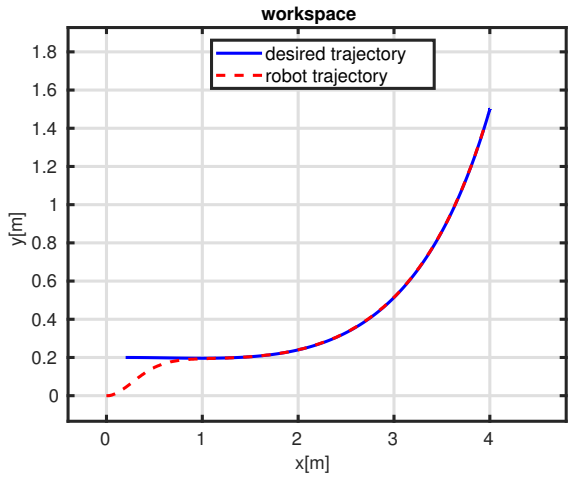

Figure 3.18 Nonlinear control experimental results : Desired and robot trajectories.

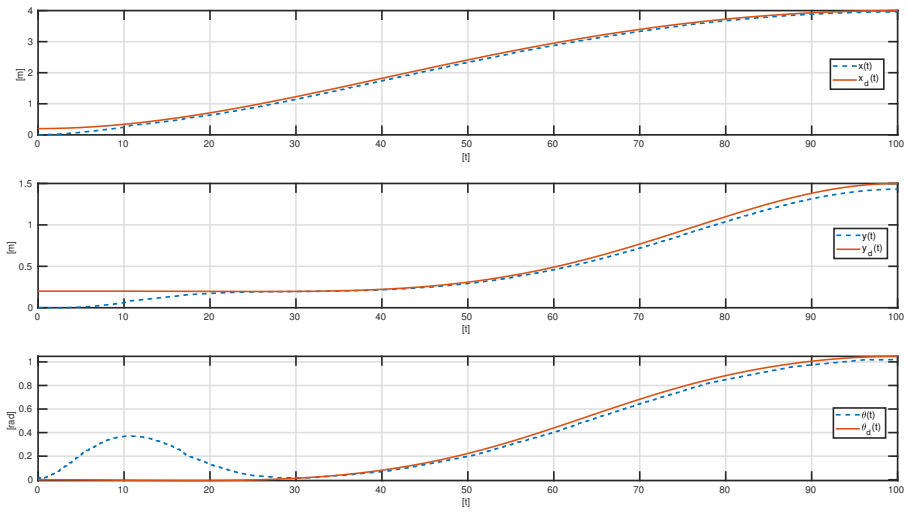

Figure 3.19 Nonlinear control experimental results : Desired and robot states.

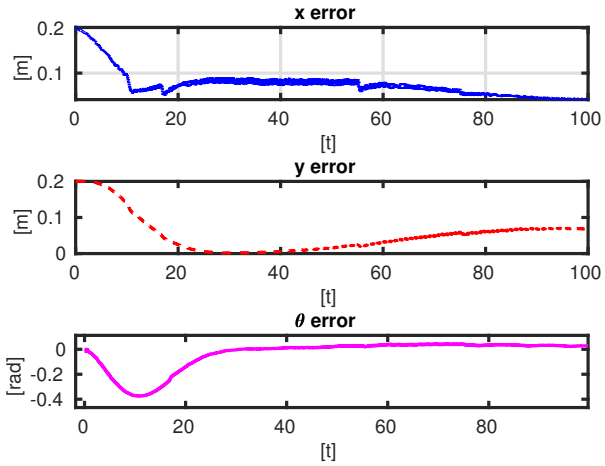

Figure 3.20 Nonlinear control experimental results : State errors

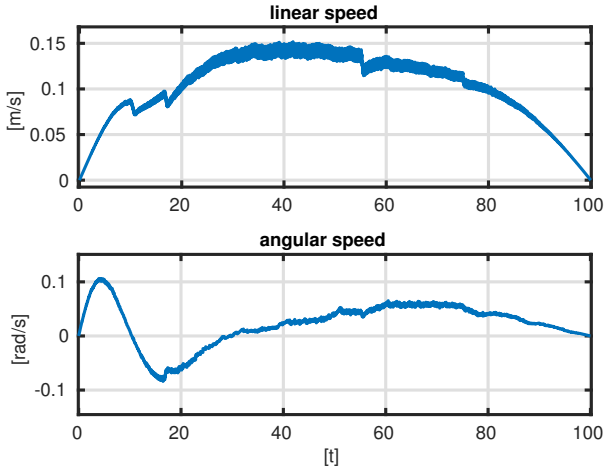

Figure 3.21 Nonlinear control experimental results : Velocities

3.4.3 Dynamic feedback linearization control experimental results

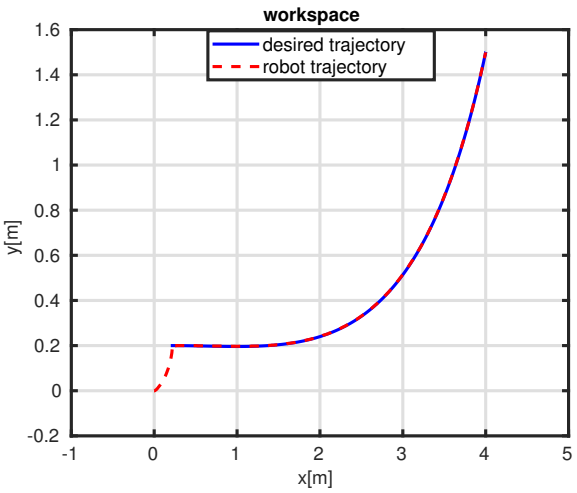

Figure 3.22 DFL experimental results : Desired and robot trajectories.

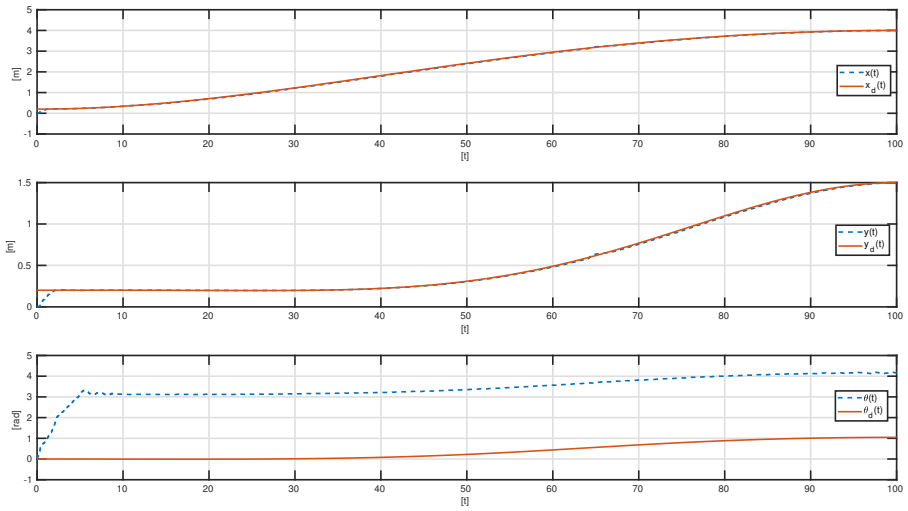

Figure 3.23 DFL experimental results : Desired and robot states.

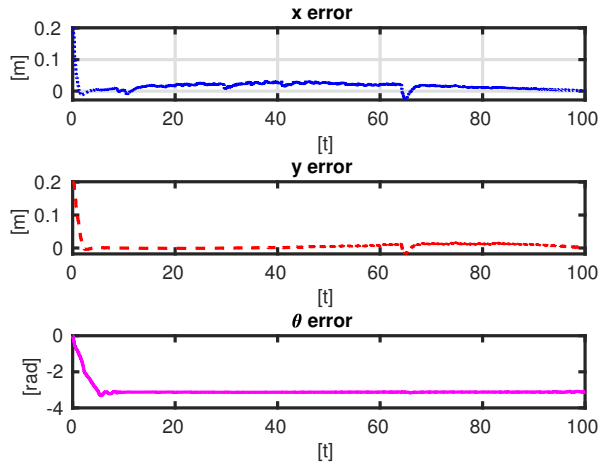

Figure 3.24 DFL experimental results : State errors

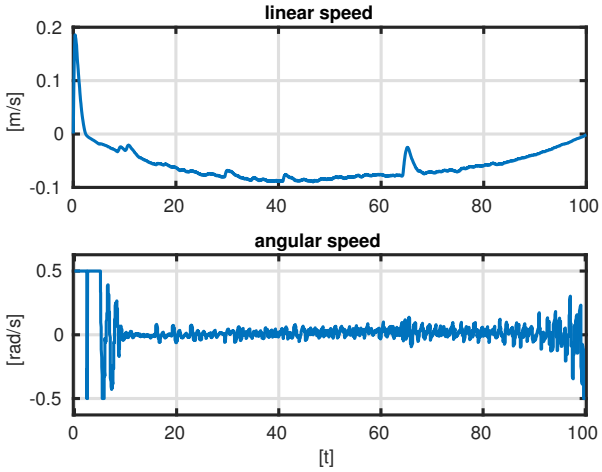

Figure 3.25 DFL experimental results : Velocities

3.5 Conclusion

As we can see from the figures, the application results are quite similar to the simulation results, which proves the accuracy of both the used model as well as the Gazebo simulator.

GENERAL CONCLUSION

With robotrs having a great impact on today's world, including many industries around the world ranging from manipulating robots which can found in automotive to pharmaceutical applications, to space exploring usages.

For the first chapter, we have introduced the kinematic model of the robot as well as the desired trajectory was planned using the cubic cartesian polynomials method, we have also discussed some of the control laws used, namely the dynamic feed linearization and the linear and non-linear controls,

For the second chapter, The fundamental concepts of ROS were brought to our attention as well as the Gazebo simulator, since the P3-DX robot is used in the in the simulation and the application, a brief discussion about it as well as its hardware.

For the third and final chapter, we have implemented the control methods that were mentioned in the first chapter, starting with the linear control and at the start of the simulation, the robot does not quite accurately track the desired trajectory but after a while it does a fairly good job of keeping track with the desired trajectory, for the non-linear control, the robot keeps track with the desired trajectory after a very short while, but the inconvenience of these previous two methods is that the robot does not reach its final configuration, for the DFL control method, the transient period of the reference tracking is a lot faster than the linear and non-linear control methods, but with that said, the θ error is not canceled, in fact practically speaking the robot accurately tracks the reference but in a backward motion, which reasoning is briefly explained in the previous chapter, it has be mentioned the results obtained from the simulation in Gazebo is extremely similar with the application results which proves the accuracy of: the used model, the Gazebo simulator as well as the ROS packages.

For future plans, four points are made :

- The first point is about solving the problem encountered with the dynamic feedback linearization.
- The second point is about adding an integrator and a derivative action to the linear and non-linear controllers to ensure that the robot reaches its final configuration.
- For the third point, other robust and sophisticated control methods can be used such as the sliding mode control, backstepping control etc .
- The fourth and final point is about using other trajectory generation methods namely the optimal trajectories method.

References

[1] 2013. Available at https://docs.ros.org/en/diamondback/api/nav_msgs/html/msg/Odometry.html.

[2] A. Allevato. Create your own urdf file -ros wiki, May 2017. Available at http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file.

[3] I. Anvari. *Non-holonomic Differential Drive Mobile Robot Control and Design : Critical Dynamics and Coupling Constraints*. PhD thesis, ARIZONA STATE UNIVERSITY, 2013.

[4] R. Belckacem and L. Ali. Robust control of wheeled mobile robots: application to vehicles type unicycle and car. Master's thesis, University of Laghouat, 2015.

[5] J. Chultz. Tf - ros wiki, 2017. Available at http://wiki.ros.org/tf.

[6] A. De Luca and M. D. Di Benedetto. Control of nonholonomic systems via dynamic compensation. *Kybernetika*, 29(6):593–608, 1993.

[7] A. De Luca, G. Oriolo, and M. Vendittelli. *Control of Wheeled Mobile Robots: An Experimental Overview*. Springer, 2002.

[8] W. Dixon, E. Zergeroglu, A. Behal, and D. M. Dawson. *Nonlinear Control of Wheeled Mobile Robots*. Springer London, 1 edition, 2001.

[9] R. Gandhinathan and L. Joseph. *ROS Robotics Projects: Build and control robots powered by the Robot Operating System, machine learning, and virtual reality*. Packt Publishing, 2 edition, 2019.

[10] R. Hedges. How to use rosaria., 2018. Available at http://wiki.ros.org/ROSARIA/Tutorials/How%20to%20use%20ROSARIA.

[11] R. Hedges. Rosaria, 2018. Available at http://wiki.ros.org/ROSARIA.

[12] A. Isidori. *Nonlinear Control Systems*. Springer, 3 edition, 1995.

[13] L. Joseph and J. Cacace. *Mastering ROS for Robotics Programming - Second Edition: Design, Build, and Simulate Complex Robots Using the Robot Operating System*. Packt Publishing, 2 edition, 2018.

[14] J. Lentin. *Ros robotics projects: Build a variety of awesome robots that can see, sense, move, and do a lot more using the powerful robot operating system*. Packt Publishing, 2017.

[15] K. M. Lynch and F. C. Park. *Modern robotics*. Cambridge University Press, 2017.

[16] S. R. Norr. *Simulation and Control of Nonholonomic Differential Drive Platforms*. PhD thesis, UNIVERSITY OF MINNESOTA, 2017.

[17] E. ORLANDO COBOS TORRES. Traction modeling and control of a differential drive mobile robot to avoid wheel slip. Master's thesis, Oklahoma State University, 2013.

[18] M. Purvis. teleop-twist-keyboard - ros wiki, 2015. Available at http://wiki.ros.org/teleop_twist_keyboard.

[19] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. 2009.

[20] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.", 2015.

[21] A. M. Robots. Pioneer 3 operations manual, 2010.

[22] A. Sears-Collins. How to publish wheel odometry information over ros – automatic addison, 2021. Available at https://automaticaddison.com/how-to-publish-wheel-odometry-information-over-ros/.

[23] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Springer London, 2009.

[24] R. L. S. Sousa, M. D. do Nascimento Forte, F. G. Nogueira, and B. C. Torrico. Trajectory tracking control of a nonholonomic mobile robot with differential drive. In *2016 IEEE Biennial Congress of Argentina (ARGENCON)*, pages 1–6. IEEE, 2016.

[25] R. Wiki. Documentation-ros wiki. *URL: http://www. ros. org*.