UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master's degree program

MASTER'S DEGREE IN INDUSTRIAL ENGINEERING

Master Thesis

DYNAMIC MODELLING, PARAMETER IDENTIFICATION, AND MOTION CONTROL OF AN OMNIDIRECTIONAL TIRE-WHEELED ROBOT

Pere Giró Perez

Supervisors:
Lluís Ros Giralt
Enric Celaya Llover

Tutor:
Maria Alba Pérez Gracia

April 2021

Dynamic modelling, parameter identification, and motion control of an omnidirectional tire-wheeled robot

by Pere Giró Perez

This work has been presented at the Universitat Politècnica de Catalunya to obtain the Master's Degree in Industrial Engineering

ETSEIB Master's degree program:
Master's Degree in Industrial Engineering

This work has been done in:
Institut de Robòtica i Informàtica Industrial, CSIC-UPC

Supervisors:
Lluís Ros Giralt
Enric Celaya Llover

Tutor:
Maria Alba Pérez Gracia

Evaluation committe:
Enrique Ernesto Zayas Figueras
M. Antonia De Los Santos López
Romà Enric Suñé Lago
Joan Puig Ortiz
Javier Ayneto Gubert

DYNAMIC MODELLING, PARAMETER IDENTIFICATION, AND MOTION CONTROL OF AN OMNIDIRECTIONAL TIRE-WHEELED ROBOT

Pere Giró Perez

Abstract

In recent years, autonomous mobile platforms are finding an increasing range of applications in inspection or surveillance tasks, or to the transport of objects, in places such as smart warehouses, factories or hospitals. In these environments it is useful for the robot to have omnidirectional capability in the plane, so it can navigate through narrow or cluttered areas, or make position and orientation changes without having to maneuver. While this capability is usually achieved with directional sliding wheels, this project studies a particular robot that achieves omnidirectionality using conventional wheels, which are easier to manufacture and maintain, and support larger loads in general. This robot, which we call the "Otbot" (for Omnidirectional tire-wheeled robot), was already conceived in the late 1990s, but all the controllers that have been proposed for it are based on purely kinematic models so far. These controllers may be sufficient if the robot is light, or if its motors are powerful, but on platforms that have to carry large loads, or that have more limited motors, it is necessary to resort to control laws based on dynamic models if the full acceleration capacities are to be exploited. This project develops a dynamic model of the Otbot, proposes a plausible methodology to identify its parameters, and designs a control law that, using this model, is able to track prescribed trajectories in an accurate and robust manner.

Acknowledgments

I would like to thank professors Lluís Ros and Enric Celaya for their great dedication and support during this last stage of the master's degree. Without their contributions and guidance, the result would be extremely different from what I am presenting today. I would also like to thank Maxime Gautier from École Polytechnique Fédérale de Lausanne - EPFL, with whom I have had the opportunity to collaborate during this project. In the same period of time he has been developing a master thesis related to mine, devoted to the planning of optimal trajectories for the Otbot. It has been of great help and support for me to have been able to share ideas and opinions with him to solve some of the different problems that have arisen throughout the work.

Contents

1 Introduction

1.1 Motivation

In recent years, autonomous mobile platforms are finding an increasing range of applications in inspection or surveillance tasks, or to the transport of objects, in places such as smart warehouses, factories or hospitals (Fig. 1.1). In these environments it is useful for the robot to have omnidirectional capability in the plane, so it can navigate through narrow corridors or cluttered areas, or make position or orientation changes without having to maneuver. While this ability is typically achieved with Mecanum wheels [1–3], in this project we study a particular robot that attains omnidirectionality using conventional wheels, which are easier to manufacture and maintain, and support larger loads in general.

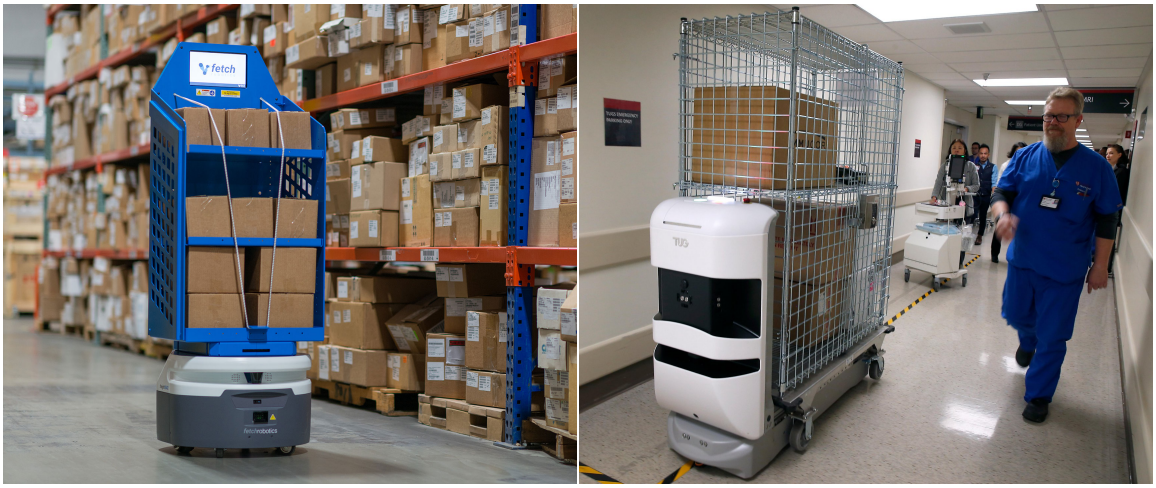


Figure 1.1: Automated material transport in a smart warehouse (left) and a hospital (right). Whereas the left robot is not omnidirectional, the one on the right achieves omnidirectionality using Mecanum wheels. Pictures courtesy of Fetch Robotics and Aethon. See https://youtu.be/KyH3xXiqbUk and https://youtu.be/MLZMAW9lqXE for details.

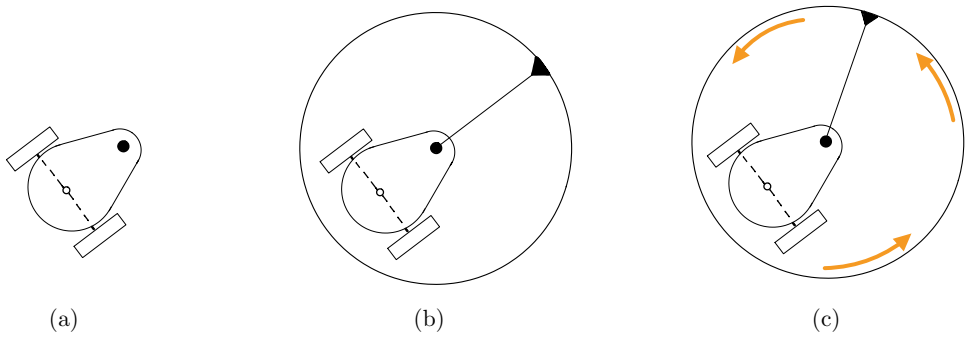


Figure 1.2: Kinematic structure of the Otbot.

This robot, which we call the "Otbot" (for "Omnidirectional tire-wheeled robot"), is composed of a rolling chassis and a circular platform mounted on top. The chassis is a classic differential drive system with two wheels actuated by DC motors (Fig. 1.2 **(a)**), and passive caster wheels for horizontal stability (not drawn). The circular platform (Fig. 1.2 **(b)**) can rotate with respect to the chassis through a pivot joint in its center, which is controlled by another DC motor (Fig. 1.2 **(c)**). The offset of the pivot joint from the wheels axis is key, as it allows the platform to undergo arbitrary 2D translations and rotations.

Such a platform was already conceived in the 1990s [4–6], but all the controllers that have been proposed for it rely only on kinematic models so far. These controllers may be sufficient if the robot is light, or if its motors are powerful, but on platforms that have to carry large loads, or that employ more limited motors, it is necessary to resort to control laws based on dynamic models if the full acceleration capacities of the robot are to be exploited. The aim of this project is to provide new tools for the analysis and use of such capacities, in situations in which the Otbot has to follow prescribed trajectories autonomously and in an accurate way.

1.2 Project objectives

Specifically, this project pursues three objectives:

1. To develop a dynamic model of the Otbot that accounts for the rolling contact constraints of its chassis and the viscous friction at its motor shafts.

2. To propose a plausible methodology to identify the dynamic parameters of this model using on-board sensors of the robot.

3. To design a control law that, using the earlier model, is able to track prescribed trajectories in a robust manner, under the presence of unmodelled disturbances.

1.3 Scope of the project

Usually, the development of system modelling and control methods is approached in two stages. In a first stage, such methods are designed and programmed, and their performance is tested in simulation. Then, in a second stage, the methods are implemented and validated in a real system. In this project we have focused on covering all tasks of the first stage, leaving those of the second for future work. Therefore, this project has been devoted to:

1. Obtain the dynamic model of the Otbot using the laws of Mechanics.
2. Propose a parameter estimation method to identify its parameters.
3. Design a control law to stabilize the system along a desired trajectory.
4. Validate the identification and control methods in simulation.

The following implementation aspects are thus left for a future project:

1. The construction of a physical protoype of the robot.
2. The development of control interfaces for its sensors and actuators.
3. The experimental validation of the methods proposed in this project.

The Otbot is an omnidirectional vehicle whose motion control is not affected by actuation singularities. However, the robot is a constrained system, as the rolling contacts of the wheels with the ground impose dependency constraints on the robot state variables. Since these constraints involve velocities and some of them cannot be integrated into a constraint involving only configuration coordinates, the robot is subject to non-holonomic constraints. This requires the use of Lagrange's equation with multipliers to obtain the dynamical model, so further manipulations have to be applied to leave this model in a form suitable for simulation and control design purposes.

Once we have the model, we will start with the identification of its dynamical parameters. We will approach this topic by means of grey-box model identification techniques. In this part, nonlinear optimization tools will be used to account for the full dynamic model. The estimation process will combine knowledge of the model written in terms of its unknown parameters, with information obtained from the on-board sensors of the robot. Since experimentation with a real prototype will not be possible in this project, we will orient our efforts to validate the parameter identification process using a simulated model of reality. Our goal will be to see if we are able

to retrieve the assumed values for the real robot parameters, under various excitation patterns and noisy sensory data.

The final goal of the project will be to design a controller able to eliminate position or velocity errors when following a trajectory. To achieve this goal we will design a computed-torque controller, which consists of a feedback linearization law combined with a standard law for the linearized system. The first law requires the inverse dynamics of the robot, which in the Otbot has a particularly simple closed-form expression. The second law is then adjusted via simple pole-placement techniques, and we apply a final verification to ensure that motor torque bounds are not surpassed under the assumed tracking errors.

We will implement all project algorithms in MATLAB, which provides the necessary means for symbolic manipulation, parameter identification, and simulation we require.

1.4 Working hypotheses

1.4.1 State sensors

Our parameter identification and trajectory control algorithms require feedback of the pivot joint angle and the absolute pose of the platform over time. While the pivot angle can be easily obtained from an angular encoder attached to the pivot motor, the estimation of the platform pose is a bit more involved. Three common approaches to obtain it involve:

1. The use of a global positioning system (GPS): the platform pose is directly provided by a GPS signal, either from an indoor or an outdoor one depending on the context.

2. The use of odometry: from the motor velocities we infer the platform velocities via forward kinematics, and then integrate such velocities to obtain the required pose.

3. The use of an inertial measurement unit (IMU): from platform acceleration and angular velocity readings we obtain the platform pose via integration.

In this project we prefer the latter approach because the use of GPS signals makes the robot too dependent on external sensors, and the use of odometry leads to poor pose estimations when wheel slippings occur on the ground. In the rest of the project, therefore, we assume the robot has an IMU sensor attached to the platform, and an angular encoder mounted at the pivot joint, which are the main sensors to be exploited by our algorithms. The only exception will be in the initial phase of our system identification procedure, which requires angular encoders on the wheels to identify the wheel moments of inertia and the viscous friction coefficients of the robot shafts.

1.4.2 Dynamic model

In order to obtain the dynamic model of the Otbot we have made the following hypotheses:

- All servomotors of the robot accept and can follow torque commands.
- The dynamic effects of the passive caster wheels can all be neglected.
- The wheels establish a perfect rolling contact with the ground, so slidings do not occur in principle.
- The robot is considered to move on flat terrain, which implies that gravitational term is null in the dynamic model.
- The only frictional forces that will be considered are those due to viscous friction of the motors. The rest of friction forces are supposed to be negligible. This includes the viscous friction of the robot against the air, the static friction at the motors, and the rolling resistance of the wheels on the ground.

1.4.3 Control system

Finally, we have assumed that the frequency at which the control loop operates is fast enough. This means that the state measurements and the evaluation of the control law are performed in a very short time, so the global system formed by the robot and its controller can be assimilated to a continuous-time system.

1.5 Document structure

The rest of the document is structured as follows:

- In Chapter 2 we explain how the kinematic model of Otbot has been obtained. This model describes the feasible motions of the robot without considering their generating forces. The resulting equations are necessary to later obtain the dynamic model of the robot.
- In Chapter 3 we obtain the dynamic model of the Otbot. By this model we mean the equations of motion that relate the accelerations of the robot to its motor torques. Both the forward and inverse dynamical problems are solved also, to later be used for simulation and control design purposes.
- In Chapter 4 we propose a grey-box model identification process to estimate the dynamical parameters of the Otbot. We explain how this process can be implemented using nonlinear regression on noisy readings from the IMU and pivot joint sensors, and validate the entire process in simulation.

- In Chapter 5 we deal with the topic of tracking control. We show how, by using feedback linearization, it is possible to design a computed-torque controller that robustly keeps the robot along a desired trajectory. We also explain how to tune this controller to stabilize the error signal of the robot in a given time.
- Finally, in Chapter 6 we provide the main conclusions of this project, and list several points for further attention.

The document has two appendices providing the MATLAB programs implementing our algorithms and simulated experiments, and the project budget.

1.6 Notation

Throughout the document we will denote scalar magnitudes with normal font, and vector ones with bold font. Vectors and matrices will be denoted with lowercase and uppercase letters, respectively. Thus, x will denote a scalar, $\mathbf{x}$ a vector, and $\mathbf{X}$ a matrix. When $\mathbf{x}$ is a vector that appears in a mathematical operation, it will be assumed to be a column vector.

When we have to make the components of a vector explicit, we will usually write them in columns and in square brackets.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \tag{1.1}$$

Sometimes, however, we will also use $\mathbf{x} = (x_1, \ldots, x_n)$ for compactness, which we will assume to be equivalent to Eq. (1.1). Similarly, we will use $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ to refer to the linear concatenation of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$. This is assumed to be equivalent to $\mathbf{x} = \left[\mathbf{x}_1^\mathsf{T}, \ldots, \mathbf{x}_n^\mathsf{T}\right]^\mathsf{T}$, where the $\mathbf{x}_i$ vectors are seen as column vectors.

To ease the explanations, we will use underbraces to define new symbols or make clarifications. For example, when we write

$$\underbrace{\text{expression}}_{\text{symbol}}$$

we mean that, thereafter, the specified symbol will denote the underbraced expression. Sometimes we will also use s_α and c_α as a shorthand for the sine and cosine of α.

2

Kinematic model

A kinematic model describes the feasible motions of a robot regardless of their generating forces and torques. This chapter starts with a description of the reference frames and state coordinates used to describe the Otbot, and then formulates the kinematic constraints imposed by the wheel contacts with the ground, as well as the solutions to the forward and inverse instantaneous kinematic problems. The resulting expressions will later be used in Chapter 3 to obtain the dynamic model for the robot. The analysis of the inverse instantaneous kinematic Jacobian will also reveal that the platform can undergo arbitrary velocities in the plane, which proves its omnidirectional capability.

2.1 Reference frames and state coordinates

To obtain the kinematic and dynamic models of the robot, we will use the reference frames shown in Fig. 2.1. The blue frame is the absolute frame fixed to the ground. The red and green frames are fixed to the chassis and the platform, respectively. Point M is the midpoint of the wheels axis and point P is the location of the pivot joint. The red axis $1'$ is always aligned with M and P. The angle between the green and red frames coincides with the pivot joint angle φ_p.

Each frame has a vector basis attached to it, whose unit vectors are directed along the frame axes. The three bases will be referred to as $B = \{1, 2, 3\}$, $B' = \{1', 2', 3'\}$ and $B'' = \{1'', 2'', 3''\}$ respectively.

The robot configuration can be described by the following six coordinates (Fig. 2.2):

- The absolute position of the pivot joint (x, y).
- The absolute angle α of the platform.
- The pivot angle φ_p between the platform and the chassis.
- The angles of the right and left wheels relative to the chassis, φ_r and φ_l.

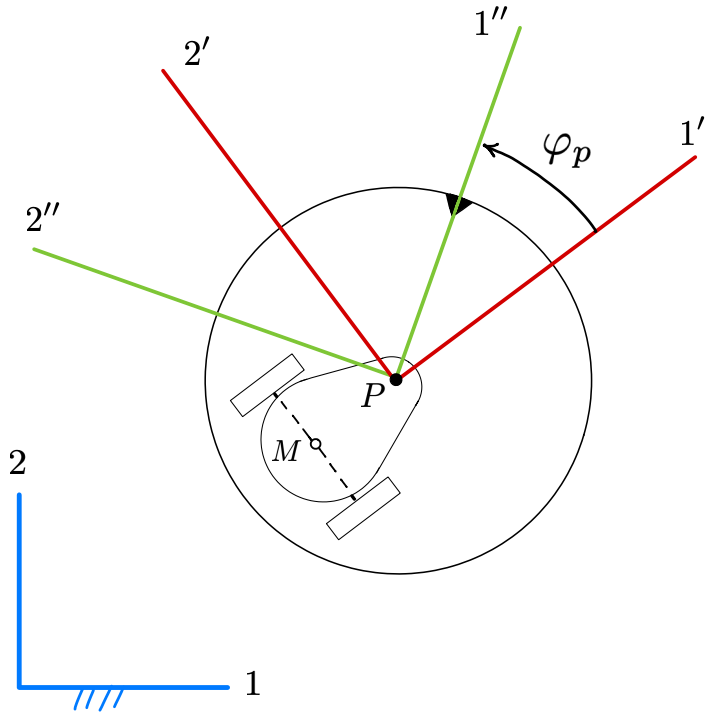


Figure 2.1: Reference frames and main nomenclature used in the models.

The robot configuration is thus given by

$$\mathbf{q} = (x, y, \alpha, \varphi_r, \varphi_l, \varphi_p), \tag{2.1}$$

so the time derivative of $\mathbf{q}$ provides the robot velocity:

$$\dot{\mathbf{q}} = (\dot{x}, \dot{y}, \dot{\alpha}, \dot{\varphi}_r, \dot{\varphi}_l, \dot{\varphi}_p).$$

Therefore, the robot state coordinates will be given by

$$\mathbf{x} = (\mathbf{q}, \dot{\mathbf{q}}) = (x, y, \alpha, \varphi_r, \varphi_l, \varphi_p, \dot{x}, \dot{y}, \dot{\alpha}, \dot{\varphi}_r, \dot{\varphi}_l, \dot{\varphi}_p).$$

2.2 Kinematic constraints

The rolling contacts of the wheels with the ground impose a kinematic constraint of the form

$$\mathbf{J}(\mathbf{q})\ \dot{\mathbf{q}} = \mathbf{0}, \tag{2.2}$$

where $\mathbf{J}(\mathbf{q})$ is a 3×6 Jacobian matrix. We derive such a constraint in two steps:

1. We first obtain the kinematic constraints imposed by the wheels.

2. We then rewrite these constraints using the $\mathbf{x}$ variables only.

We next see these steps in detail. In our derivations, we use $\mathbf{v}(Q)$ to denote the velocity of a point Q of the robot. The basis in which $\mathbf{v}(Q)$ is expressed will be mentioned explicitly, or understood by context.

2.2.1 Kinematic constraints of the differential drive

For the moment, let us neglect the platform and focus our attention on the chassis, as seen in Fig. 2.3. The chassis pose is given by the position vector (a, b) of point M, and by the orientation angle θ. We use l_1 and l_2 to refer to the pivot offset from M and the half-length of the wheels axis, respectively. Also, C_r and C_l denote the centers of the right and left wheels, and P_r and P_l are the contact points of the wheels with the ground. The two wheels have the same radius r.

The rolling contact constraints of the chassis can be found by computing the velocities $\mathbf{v}(P_r)$ and $\mathbf{v}(P_l)$ in terms of $\dot{a}$, $\dot{b}$, $\dot{\theta}$, $\dot{\varphi}_r$, and $\dot{\varphi}_l$ (i.e., as if the robot were a floating kinematic tree) and forcing these velocities to be zero (as the wheels do not slide when placed on the ground).

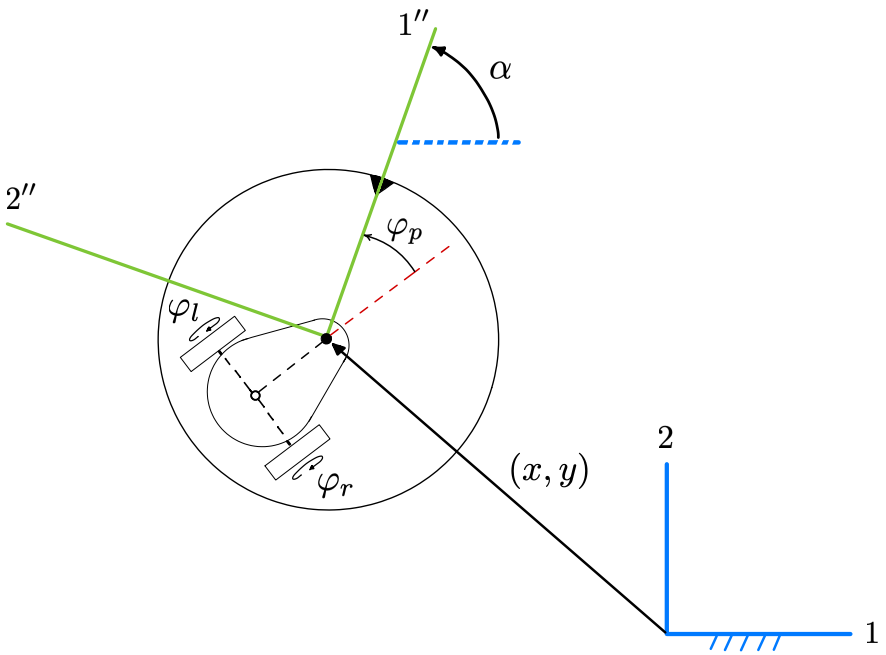


Figure 2.2: Otbot's Configuration and state coordinates.

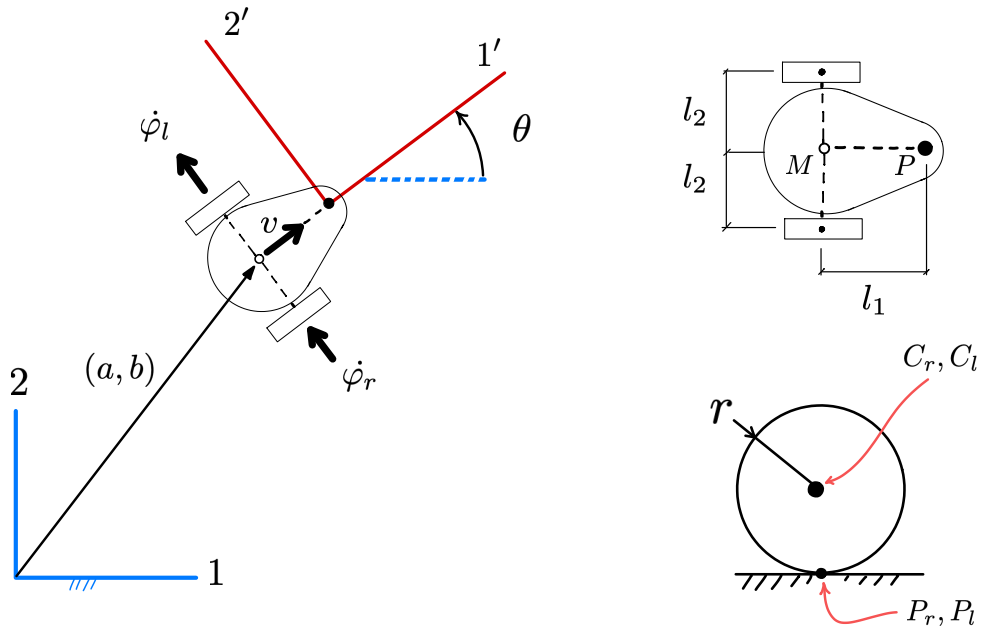


Figure 2.3: Notation used in the kinematic modelling of the Otbot chassis.

To obtain $\mathbf{v}(P_r)$ and $\mathbf{v}(P_l)$, note first that the velocity of M can only be directed along axis $1'$, since lateral slipping is forbidden under perfect rolling. Therefore in $B' = \{1', 2', 3'\}$ we have

$$\mathbf{v}(M) = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix}.$$

Also note that, if $\boldsymbol{\omega}_{\text{chassis}}$ is the angular velocity of the chassis, we have

$$\mathbf{v}(C_r) = \mathbf{v}(M) + \boldsymbol{\omega}_{\text{chassis}} \times \mathbf{MC}_r = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ -l_2 \\ 0 \end{bmatrix} = \begin{bmatrix} v + l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{v}(C_l) = \mathbf{v}(M) + \boldsymbol{\omega}_{\text{chassis}} \times \mathbf{MC}_l = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ l_2 \\ 0 \end{bmatrix} = \begin{bmatrix} v - l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix}.$$

The velocities of the ground contact points are thus given by

$$\mathbf{v}(P_r) = \mathbf{v}(C_r) + \boldsymbol{\omega}_{\text{wheel}} \times \mathbf{C}_r\mathbf{P}_r = \begin{bmatrix} v + l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \dot{\varphi}_r \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -r \end{bmatrix} = \begin{bmatrix} v + l_2\,\dot{\theta} - r\,\dot{\varphi}_r \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{v}(P_l) = \mathbf{v}(C_l) + \boldsymbol{\omega}_{\text{wheel}} \times \mathbf{C}_l\mathbf{P}_l = \begin{bmatrix} v - l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \dot{\varphi}_l \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -r \end{bmatrix} = \begin{bmatrix} v - l_2\,\dot{\theta} - r\,\dot{\varphi}_l \\ 0 \\ 0 \end{bmatrix}.$$

Since P_r and P_l do not slip, $\mathbf{v}(P_r)$ and $\mathbf{v}(P_l)$ must be zero, which gives the two fundamental constraints of the robot:

$$v + l_2\dot{\theta} - r\dot{\varphi}_r = 0$$

$$v - l_2\dot{\theta} - r\dot{\varphi}_l = 0.$$

It is now easy to solve for v and $\dot{\theta}$ in these two equations:

$$v = \frac{r\,(\dot{\varphi}_l + \dot{\varphi}_r)}{2} \tag{2.3}$$

$$\dot{\theta} = -\frac{r\,(\dot{\varphi}_l - \dot{\varphi}_r)}{2l_2} \tag{2.4}$$

Note from Fig. 2.3 that in the basis $B = \{1, 2, 3\}$

$$\mathbf{v}(M) = \begin{bmatrix} \dot{a} \\ \dot{b} \\ 0 \end{bmatrix},$$

and that it must be

$$\begin{cases} \dot{a} = v \cdot \cos\theta \\ \dot{b} = v \cdot \sin\theta \end{cases}$$

By substituting Eq. (2.3) in these two equations, and also considering Eq. (2.4), we obtain

$$\begin{cases} \dot{a} = \cos(\theta)\left(\dfrac{r\dot{\varphi}_l}{2} + \dfrac{r\dot{\varphi}_r}{2}\right) \\ \dot{b} = \sin(\theta)\left(\dfrac{r\dot{\varphi}_l}{2} + \dfrac{r\dot{\varphi}_r}{2}\right) \\ \dot{\theta} = -\dfrac{r\,(\dot{\varphi}_l - \dot{\varphi}_r)}{2l_2} \end{cases} \tag{2.5}$$

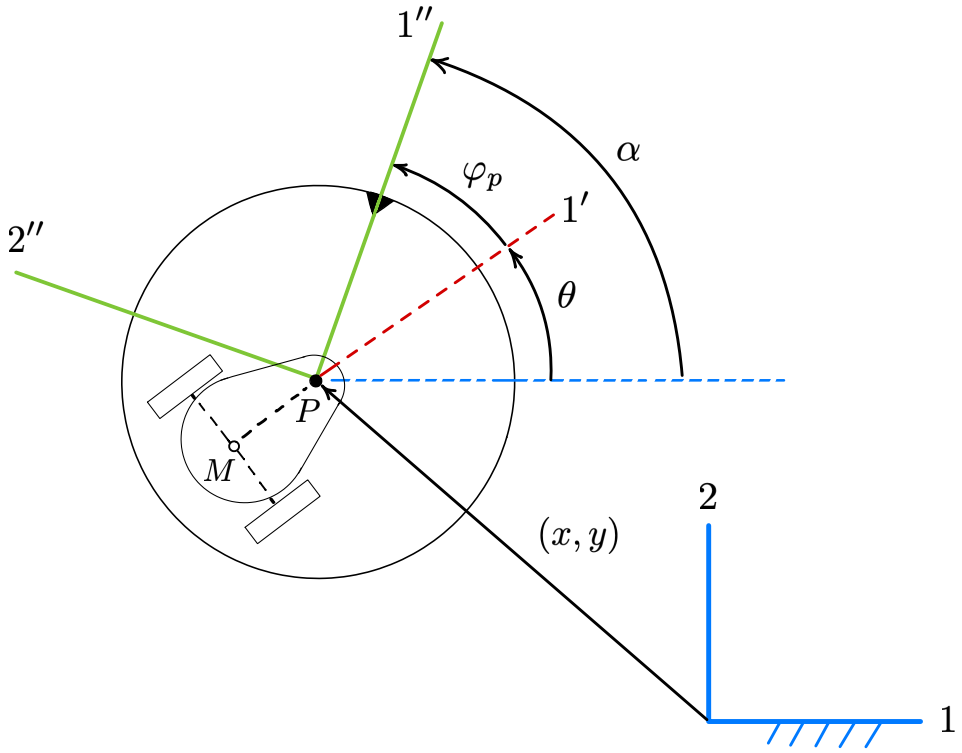


Figure 2.4: Relationship between α, φ_p, and θ.

This system provides the forward kinematic solution for the differential drive. It can also be viewed as a system of kinematic constraints that expresses the rolling contact constraints of the chassis.

2.2.2 Kinematic constraints of the whole robot

To obtain the kinematic constraints of Otbot itself, we just need to rewrite the previous system using the $\mathbf{x}$ variables only. This entails substituting θ, $\dot{\theta}$, $\dot{a}$, and $\dot{b}$ by their expressions in terms of $\mathbf{x}$ coordinates.

From Fig. 2.4 it is clear that

$$\theta = \alpha - \varphi_p$$
$$\dot{\theta} = \dot{\alpha} - \dot{\varphi}_p$$

and we also see that

$$\mathbf{v}(M) = \mathbf{v}(P) + \omega_{\text{chassis}} \times \mathbf{PM},$$

so using $B = \{1, 2, 3\}$ we can write

$$\begin{bmatrix} \dot{a} \\ \dot{b} \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} -l_1 \cdot \cos\theta \\ -l_1 \cdot \sin\theta \\ 0 \end{bmatrix}$$
$$= \begin{bmatrix} \dot{x} + \dot{\theta}\, l_1 \sin\theta \\ \dot{y} - \dot{\theta}\, l_1 \cos\theta \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} + (\dot{\alpha} - \dot{\varphi}_p)\, l_1\ \sin(\alpha - \varphi_p) \\ \dot{y} - (\dot{\alpha} - \dot{\varphi}_p)\, l_1\ \cos(\alpha - \varphi_p) \\ 0 \end{bmatrix}.$$

In sum we have the relationships

$$\begin{cases} \theta = \alpha - \varphi_p \\ \dot{\theta} = \dot{\alpha} - \dot{\varphi}_p \\ \dot{a} = \dot{x} + (\dot{\alpha} - \dot{\varphi}_p)\, l_1\ \sin(\alpha - \varphi_p) \\ \dot{b} = \dot{y} - (\dot{\alpha} - \dot{\varphi}_p)\, l_1\ \cos(\alpha - \varphi_p) \end{cases}$$

which give the desired values of θ, $\dot{\theta}$, $\dot{a}$, and $\dot{b}$ in terms of $\mathbf{q}$ and $\dot{\mathbf{q}}$. We thus can substitute these expressions in Eq. (2.5) to obtain:

$$\dot{x} + l_1 \sin(\alpha - \varphi_p)\ (\dot{\alpha} - \dot{\varphi}_p) = \cos(\alpha - \varphi_p)\ \left(\frac{r\dot{\varphi}_l}{2} + \frac{r\dot{\varphi}_r}{2}\right), \tag{2.6}$$

$$\dot{y} - l_1 \cos(\alpha - \varphi_p)\ (\dot{\alpha} - \dot{\varphi}_p) = \sin(\alpha - \varphi_p)\ \left(\frac{r\dot{\varphi}_l}{2} + \frac{r\dot{\varphi}_r}{2}\right), \tag{2.7}$$

$$\dot{\alpha} - \dot{\varphi}_p = -\frac{r\,(\dot{\varphi}_l - \dot{\varphi}_r)}{2\,l_2}. \tag{2.8}$$

These are the *kinematic constraints* of the Otbot expressed in $\mathbf{x}$ coordinates. Additionally, by re-ordering and integrating the last equation of this system, we can obtain an holonomic constraint of our system

$$\alpha - \frac{r}{2\ell_2}\varphi_r + \frac{r}{2\ell_2}\varphi_e - \varphi_p + K_0 = 0, \tag{2.9}$$

where $K_0 = \alpha(0) - \frac{r}{2\ell_2}\varphi_r(0) + \frac{r}{2\ell_2}\varphi_e(0) - \varphi_p(0)$. This constraint directly relates the different angles in the model.

In matrix form, Eqs. (2.6) - (2.8) can be written as

$$\mathbf{J}\left(\mathbf{q}\right)\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\alpha}\\ \dot{\varphi}_r\\ \dot{\varphi}_l\\ \dot{\varphi}_p\end{bmatrix}=\mathbf{0},$$

where

$$\mathbf{J}(\mathbf{q})=\begin{bmatrix}1 & 0 & l_1 s_{\alpha-\varphi_p} & -\frac{1}{2}rc_{\alpha-\varphi_p} & -\frac{1}{2}rc_{\alpha-\varphi_p} & -l_1 s_{\alpha-\varphi_p}\\ 0 & 1 & -l_1 c_{\alpha-\varphi_p} & -\frac{1}{2}rs_{\alpha-\varphi_p} & -\frac{1}{2}rs_{\alpha-\varphi_p} & l_1 c_{\alpha-\varphi_p}\\ 0 & 0 & 1 & -\frac{r}{2l_2} & \frac{r}{2l_2} & -1\end{bmatrix} \tag{2.10}$$

2.3 Solution to the instantaneous kinematic problems

The configuration $\mathbf{q}$ in Eq. (2.1) can be separated into two vectors $\mathbf{p} = (x, y, \alpha)$ and $\boldsymbol{\varphi} = (\varphi_r, \varphi_l, \varphi_p)$. These vectors provide the task and joint space coordinates of the robot. Their time derivatives $\dot{\boldsymbol{\varphi}}$ and $\dot{\mathbf{p}}$ are called the motor speeds and the platform twist, respectively. In this section, we solve the following two problems:

- Forward Instantaneous Kinematic Problem (FIKP): Obtain $\dot{\mathbf{p}}$ as a function of $\dot{\boldsymbol{\varphi}}$.
- Inverse Instantaneous Kinematic Problem (IIKP): Obtain $\dot{\boldsymbol{\varphi}}$ as a function of $\dot{\mathbf{p}}$.

It is also shown that the IIKP is always solvable irrespective of the robot configuration. This implies that the platform of the Otbot can move omnidirectionally in the plane.

2.3.1 Forward problem

We only have to isolate $\dot{x}$, $\dot{y}$, and $\dot{\alpha}$ from the earlier Eqs. (2.6) - (2.8): defining $\mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} = \mathbf{0}$:

$$\dot{x}=\frac{l_2\, r\dot{\varphi}_l\cos\left(\alpha-\varphi_p\right)+l_2\, r\dot{\varphi}_r\cos\left(\alpha-\varphi_p\right)+l_1\, r\dot{\varphi}_l\sin\left(\alpha-\varphi_p\right)-l_1\, r\dot{\varphi}_r\sin\left(\alpha-\varphi_p\right)}{2\, l_2}$$

$$\dot{y}=\frac{l_1\, r\dot{\varphi}_r\cos\left(\alpha-\varphi_p\right)-l_1\, r\dot{\varphi}_l\cos\left(\alpha-\varphi_p\right)+l_2\, r\dot{\varphi}_l\sin\left(\alpha-\varphi_p\right)+l_2\, r\dot{\varphi}_r\sin\left(\alpha-\varphi_p\right)}{2\, l_2}$$

$$\dot{\alpha}=\frac{2\, l_2\dot{\varphi}_p-r\dot{\varphi}_l+r\dot{\varphi}_r}{2\, l_2}$$

These equations directly provide $\dot{\mathbf{p}}$ as a function of $\dot{\boldsymbol{\varphi}}$. This function can be expressed as

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \end{bmatrix} = \mathbf{M}_{FIK}(\mathbf{q}) \begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix},$$

where

$$\mathbf{M}_{FIK}(\mathbf{q}) = \begin{bmatrix} \frac{l_2\, r\cos(\alpha-\varphi_p)-l_1\, r\sin(\alpha-\varphi_p)}{2\, l_2} & \frac{l_2\, r\cos(\alpha-\varphi_p)+l_1\, r\sin(\alpha-\varphi_p)}{2\, l_2} & 0 \\ \frac{l_1\, r\cos(\alpha-\varphi_p)+l_2\, r\sin(\alpha-\varphi_p)}{2\, l_2} & -\frac{l_1\, r\cos(\alpha-\varphi_p)-l_2\, r\sin(\alpha-\varphi_p)}{2\, l_2} & 0 \\ \frac{r}{2\, l_2} & -\frac{r}{2\, l_2} & 1 \end{bmatrix}$$

2.3.2 Inverse problem

We now wish to find $\dot{\boldsymbol{\varphi}}$ as a function of $\dot{\mathbf{p}}$. Clearly, this function is given by

$$\begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix} = \mathbf{M}_{IIK}(\mathbf{q}) \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \end{bmatrix},$$

where

$$\mathbf{M}_{IIK}(\mathbf{q}) = \mathbf{M}_{FIK}^{-1}(\mathbf{q}) = \begin{bmatrix} \frac{l_1\cos(\alpha-\varphi_p)-l_2\sin(\alpha-\varphi_p)}{l_1\, r} & \frac{l_2\cos(\alpha-\varphi_p)+l_1\sin(\alpha-\varphi_p)}{l_1\, r} & 0 \\ \frac{l_1\cos(\alpha-\varphi_p)+l_2\sin(\alpha-\varphi_p)}{l_1\, r} & -\frac{l_2\cos(\alpha-\varphi_p)-l_1\sin(\alpha-\varphi_p)}{l_1\, r} & 0 \\ \frac{\sin(\alpha-\varphi_p)}{l_1} & -\frac{\cos(\alpha-\varphi_p)}{l_1} & 1 \end{bmatrix}$$

Note that the determinant of $\mathbf{M}_{FIK}(\mathbf{q})$ is

$$\det(\mathbf{M}_{FIK}(\mathbf{q})) = -\frac{l_1\, r^2}{2\, l_2}.$$

Since r, l_1, and l_2 are all positive in Otbot, we see that $\mathbf{M}_{FIK}^{-1}(\mathbf{q})$ always exists, so the IIKP is solvable in all configurations of the robot. An important consequence of this fact is that the platform will be able to move under any twist $\dot{\mathbf{p}}$ in the plane. In other words, it will be an omnidirectional platform.

2.4 The kinematic model in control form

The following equations

$$\begin{cases} \dot{\mathbf{p}} = \mathbf{M}_{FIK}(\mathbf{q})\ \dot{\boldsymbol{\varphi}} \\ \dot{\boldsymbol{\varphi}} = \dot{\boldsymbol{\varphi}} \end{cases}$$

can be written in the usual form used in control engineering,

$$\dot{\mathbf{q}} = \mathbf{f}_{\text{kin}}(\mathbf{q}, \boldsymbol{\omega}),$$

where $\boldsymbol{\omega} = \dot{\boldsymbol{\varphi}}$ and

$$\mathbf{f}_{\text{kin}}(\mathbf{q}, \boldsymbol{\omega}) = \begin{bmatrix} \mathbf{M}_{FIK}(\mathbf{q}) \\ \mathbf{I}_3 \end{bmatrix} \boldsymbol{\omega}.$$

Notice that in this model $\boldsymbol{\omega}$ plays the role of the control actions, which are motor velocities in this case.

If necessary, we could use this model for controlling the Otbot motions. In doing so, we would neglect the system dynamics, but a strong point of this model is its simplicity, which leads to higher frequency controllers. Moreover, the model would be sufficient if the commanded velocities $\boldsymbol{\omega}(t)$ where easy to control (e.g., if $\boldsymbol{\omega}(t)$ were smooth enough for the motors at hand). However, our interest is in obtaining a more complete and accurate model so we can simulate and control our robot as realistically as possible, taking into account the system dynamics. We develop such a model in the following chapter.

3
Dynamic model

In this chapter we obtain the dynamic model of the Otbot, which provides the relationship between the accelerations of the system and the torques applied by its motors. Since the Otbot is a non-holonomic system, the model will be derived using Lagrange's equations with multipliers [7]. After obtaining all terms in this equation, we will manipulate it to solve the inverse and forward dynamics of the robot. The obtained solutions and their related equations will be needed for simulation and control design purposes in our implementations. The forward dynamics solution, in particular, yields the dynamic model in control form, $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$, which is the form typically assumed by many of the algorithms we use.

3.1 Lagrange's equation with multipliers

It is well known that, in an inertial reference frame $\mathcal{F}$, the equation of motion of a non-holonomic system can be written as

$$\mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}(\mathbf{q})^{\mathsf{T}}\,\boldsymbol{\lambda} = \mathbf{Q}_a + \mathbf{Q}_f, \tag{3.1}$$

where

- $\mathbf{q}$ is the configuration vector of the robot.
- $\mathbf{M}(\mathbf{q})$ is the mass matrix characterizing the kinetic energy of the system in $\mathcal{F}$.
- $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ is the generalized Coriolis and centrifugal force matrix.
- $\mathbf{G}(\mathbf{q}) = \partial U / \partial \mathbf{q}$, where $U(\mathbf{q})$ is the potential energy function of the robot.
- $\mathbf{J}(\mathbf{q})$ is the constraint Jacobian of the robot.

- $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers.
- $\mathbf{Q}_a$ is the generalized force of actuation, which models the motor forces.
- $\mathbf{Q}_f$ is the generalized force modelling all friction forces in the system.

To set up this equation for the Otbot we let $\mathcal{F}$ be the absolute frame fixed to the ground (Section 2.1), which is assumed to be inertial at all effects. Recall that in this frame the robot configuration is given by

$$\mathbf{q} = (x, y, \alpha, \varphi_r, \varphi_l, \varphi_p)\,.$$

We also define $\mathbf{u}$ as

$$\mathbf{u} = (\tau_r, \tau_l, \tau_p)\,,$$

where τ_r, τ_l, and τ_p are the torques applied by the motors to the left and right wheels and to the pivot joint, respectively. Note that $\mathbf{G}(\mathbf{q}) = \mathbf{0}$ in our case, as the Otbot moves on flat terrain, and that $\mathbf{J}(\mathbf{q})$ is the Jacobian in Eq. (2.10). We next see how to obtain the remaining terms in Eq. (3.1): $\mathbf{M}(\mathbf{q})$, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$, $\mathbf{Q}_a$, and $\mathbf{Q}_f$.

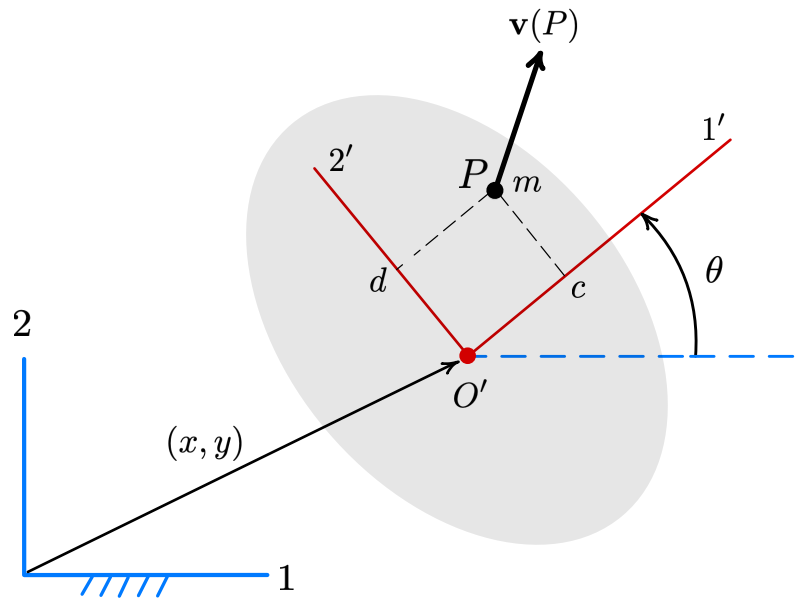


Figure 3.1: Notation used to derive a general expression for the translational kinetic energy of a point of mass m located at a point P of a body moving under planar motion (shown in grey). The velocity of the body is given by $\dot{x}$, $\dot{y}$ and $\dot{\theta}$. The coordinates of P in the body-fixed frame $\{1', 2'\}$ are (c, d).

3.1.1 Mass matrix

The mass matrix $\mathbf{M}(\mathbf{q})$ describes the mass of our entire dynamical system in terms of the configuration variables. This matrix can be computed by writing the total kinetic energy T of the robot as a function of $\mathbf{q}$ and $\dot{\mathbf{q}}$, and expressing T in the form

$$T(\mathbf{q}, \dot{\mathbf{q}}) = \frac{1}{2}\, \dot{\mathbf{q}}^{\mathsf{T}}\, \mathbf{M}(\mathbf{q})\, \dot{\mathbf{q}}.$$

The robot has four bodies: the main body of the chassis, the left wheel, the right wheel, and the platform. We have to compute the translational and rotational kinetic energies of each body and add them all to obtain the expression of T for the complete system.

To compute the translational kinetic energy, we first derive a general expression for the kinetic energy T_t of a point mass m located at some point P of a body under planar motion (see Fig. 3.1). We wish to express T_t in terms of the velocity coordinates of the body, $\mathbf{w} = (\dot{x}, \dot{y}, \dot{\theta})$, the coordinates of P in the body-fixed frame $\{O', 1', 2'\}$, the body angle θ, and m. To obtain T_t, we first express the velocity of P as

$$\mathbf{v}(P) = \mathbf{v}(O') + \boldsymbol{\omega}_{\text{body}} \times \mathbf{O'P},$$

where $\boldsymbol{\omega}_{\text{body}} = (0, 0, \dot{\theta})$. This yields

$$\begin{bmatrix} v_1 \\ v_2 \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} c\cos\theta - d\sin\theta \\ c\sin\theta + d\cos\theta \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} - c\dot{\theta}\sin\theta - d\dot{\theta}\cos\theta \\ \dot{y} + c\dot{\theta}\cos\theta - d\dot{\theta}\sin\theta \\ 0 \end{bmatrix}.$$

By adding $\dot{\theta}$ in the third row, we then obtain

$$\underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \dot{\theta} \end{bmatrix}}_{\mathbf{t}} = \underbrace{\begin{bmatrix} 1 & 0 & -c\sin\theta - d\cos\theta \\ 0 & 1 & c\cos\theta - d\sin\theta \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}}_{\mathbf{w}}.$$

Now note that T_t can be expressed as follows in terms of $\mathbf{t}$:

$$T_t = \frac{1}{2}m(v_1^2 + v_2^2) = \frac{1}{2} \underbrace{\begin{bmatrix} v_1 & v_2 & \dot{\theta} \end{bmatrix}}_{\mathbf{t}^{\mathsf{T}}} \underbrace{\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\mathbf{M}'} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \dot{\theta} \end{bmatrix}}_{\mathbf{t}}.$$

Using $\mathbf{t} = \mathbf{A}\ \mathbf{w}$, we can now express T_t as

$$T_t = \frac{1}{2}\ \mathbf{w}^\mathsf{T}\ \underbrace{\mathbf{A}^\mathsf{T}\mathbf{M}'\mathbf{A}}_{\mathbf{M}_t}\mathbf{w} = \frac{1}{2}\ \mathbf{w}^\mathsf{T}\ \mathbf{M}_t\ \mathbf{w},$$

where

$$\mathbf{M}_t(c,d,m) = \mathbf{A}\ ^\mathsf{T}\mathbf{M}'\ \mathbf{A} = \\ = \begin{bmatrix} m & 0 & -c\ m\sin\theta - d\ m\cos\theta \\ 0 & m & c\ m\cos\theta - d\ m\sin\theta \\ -c\ m\sin\theta - d\ m\cos\theta & c\ m\cos\theta - d\ m\sin\theta & m(c^2+d^2) \end{bmatrix}.$$

Using the relations above, we thus can express the translational kinetic energy of a point mass m located at some point P of a body under planar motion as:

$$T_t(c,d,m,\dot{x},\dot{y},\theta,\dot{\theta}) = \frac{1}{2}\ \mathbf{w}^\mathsf{T}\ \mathbf{M}_t(c,d,m)\ \mathbf{w}.$$

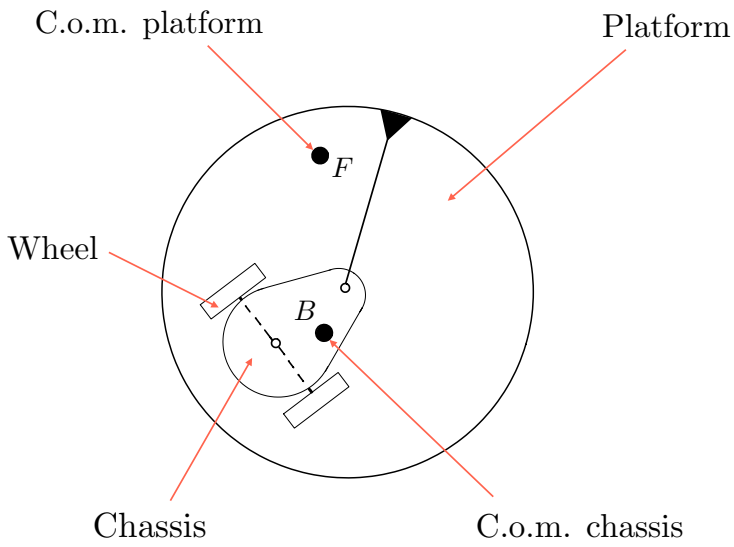
Using this formula and the dynamic parameters defined in Table 3.1, it is easy to see that the translational kinetic energy of the rolling chassis (including the wheels) and the platform are given by

$$T_{\text{trans,chass}} = T_t(x_B, y_B, m_c, \dot{x}, \dot{y}, \theta, \dot{\theta})$$
$$T_{\text{trans,plat}} = T_t(x_F, y_F, m_p, \dot{x}, \dot{y}, \alpha, \dot{\alpha})$$

Again with the same parameters, the rotational kinetic energy of the chassis, the right and left wheels, and the platform can be expressed as

$$T_{\text{rot,chass}} = \frac{1}{2}\cdot I_c \cdot \dot{\theta}^2,$$
$$T_{\text{rot,r}} = \frac{1}{2}\cdot I_a \cdot \dot{\varphi}_r^2,$$
$$T_{\text{rot,l}} = \frac{1}{2}\cdot I_a \cdot \dot{\varphi}_l^2,$$
$$T_{\text{rot,plat}} = \frac{1}{2}\cdot I_p \cdot \dot{\alpha}^2,$$

where we emphasize that I_c is the vertical moment of inertia of the rolling chassis about its center of mass B (i.e., assuming that the chassis body and the wheels form a single rigid body).



Symbol	Meaning
(x_B, y_B)	Coordinates of the center of mass of the chassis in the chassis frame
(x_F, y_F)	Coordinates of the center of mass of the platform in the platform frame
m_c	Total mass of the rolling chassis
m_p	Mass of the platform
I_c	Vertical moment of inertia of the rolling chassis at point B
I_p	Vertical moment of inertia of the platform at point F
I_a	Axial moment of inertia of a wheel

Table 3.1: Robot parameters and their meaning.

The total kinetic energy of the robot is thus given by

$$T(\mathbf{q}, \dot{\mathbf{q}}) = T_{\text{trans,chass}} + T_{\text{trans,plat}} + T_{\text{rot,chass}} + T_{\text{rot,r}} + T_{\text{rot,l}} + T_{\text{rot,plat}},$$

Since $T(\mathbf{q}, \dot{\mathbf{q}})$ is a quadratic form, the Hessian of T will provide the desired mass matrix $\mathbf{M}(\mathbf{q})$. With the help of a symbolic manipulation tool (Appendix A) we obtain the expression shown in Fig. 3.2.

3.1.2 Coriolis matrix

To compute the Coriolis matrix $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ we have used the expression

$$\mathbf{C}_{\text{ij}} = \frac{1}{2} \sum_{\text{k=1}}^{\text{n}} \left(\frac{\partial \mathbf{M}_{\text{ij}}}{\partial \mathbf{q}_{\text{k}}} + \frac{\partial \mathbf{M}_{\text{ik}}}{\partial \mathbf{q}_{\text{j}}} - \frac{\partial \mathbf{M}_{\text{kj}}}{\partial \mathbf{q}_{\text{i}}} \right) \dot{\mathbf{q}}_{\text{k}} \tag{3.2}$$

$$\mathbf{M}(\mathbf{q}) = \begin{bmatrix}
m_c + m_p & 0 & \begin{matrix}-m_p\, y_F\, c_\alpha \\ -m_p\, x_F\, s_\alpha \\ -m_c\, y_B\, c_\theta \\ -m_c\, x_B\, s_\theta\end{matrix} & 0 & 0 & \begin{matrix}m_c\, y_B\, c_\theta \\ +m_c\, x_B\, s_\theta\end{matrix} \\
0 & m_c + m_p & \begin{matrix}m_p\, x_F\, c_\alpha \\ -m_p\, y_F\, s_\alpha \\ +m_c\, x_B\, c_\theta \\ -m_c\, y_B\, s_\theta\end{matrix} & 0 & 0 & \begin{matrix}m_c\, y_B\, s_\theta \\ -m_c\, x_B\, c_\theta\end{matrix} \\
\begin{matrix}-m_p\, y_F\, c_\alpha \\ -m_p\, x_F\, s_\alpha \\ -m_c\, y_B\, c_\theta \\ -m_c\, x_B\, s_\theta\end{matrix} & \begin{matrix}m_p\, x_F\, c_\alpha \\ -m_p\, y_F\, s_\alpha \\ +m_c\, x_B\, c_\theta \\ -m_c\, y_B\, s_\theta\end{matrix} & \begin{matrix}m_c\, {x_B}^2 + m_p\, {x_F}^2 \\ +m_c\, {y_B}^2 + m_p\, {y_F}^2 \\ +I_c + I_p\end{matrix} & 0 & 0 & \begin{matrix}-m_c\, {x_B}^2 \\ -m_c\, {y_B}^2 \\ -I_c\end{matrix} \\
0 & 0 & 0 & I_a & 0 & 0 \\
0 & 0 & 0 & 0 & I_a & 0 \\
\begin{matrix}m_c\, y_B\, c_\theta \\ +m_c\, x_B\, s_\theta\end{matrix} & \begin{matrix}m_c\, y_B\, s_\theta \\ -m_c\, x_B\, c_\theta\end{matrix} & \begin{matrix}-m_c\, {x_B}^2 \\ -m_c\, {y_B}^2 \\ -I_c\end{matrix} & 0 & 0 & \begin{matrix}m_c\, {x_B}^2 \\ +m_c\, {y_B}^2 \\ +I_c\end{matrix}
\end{bmatrix}$$

Figure 3.2: Symbolic expression of the mass matrix $\mathbf{M}(\mathbf{q})$

given in [7], where $\mathbf{M}_{ij}$ and $\mathbf{C}_{ij}$ refer to the (i, j) entry of $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ and $\mathbf{M}(\mathbf{q})$ respectively. This yields the expression for $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ shown in Fig. 3.3.

3.1.3 Generalized force of actuation

To write the expression for the generalized actuation force $\mathbf{Q}_a$ we will use the method of virtual power. This requires calculating the power P_a generated by the actuation forces $\mathbf{u} = (\tau_r, \tau_l, \tau_p)$ under a virtual velocity $\dot{\mathbf{q}}$ at which the mechanism is moving, and writing it in the form

$$P_a = \mathbf{Q}_a^\mathsf{T}\, \dot{\mathbf{q}}. \tag{3.3}$$

Notice that, clearly,

$$P_a = \underbrace{\begin{bmatrix} \tau_r & \tau_l & \tau_p \end{bmatrix}}_{\mathbf{u}^\mathsf{T}} \underbrace{\begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix}}_{\dot{\boldsymbol{\varphi}}}$$

$$\mathbf{C}(\mathbf{q},\dot{\mathbf{q}}) = \begin{bmatrix} 0 & 0 & \begin{matrix}-\dot\theta\, m_c(x_B\, c_\theta - y_B\, s_\theta)\\ -\dot\alpha\, m_p(x_F\, c_\alpha - y_F\, s_\alpha)\end{matrix} & 0 & 0 & \dot\theta\, m_c\,(x_B\, c_\theta - y_B\, s_\theta) \\ 0 & 0 & \begin{matrix}-\dot\theta\, m_c(y_B\, c_\theta + x_B\, s_\theta)\\ -\dot\alpha\, m_p(y_F\, c_\alpha + x_F\, s_\alpha)\end{matrix} & 0 & 0 & \dot\theta\, m_c\,(y_B\, c_\theta + x_B\, s_\theta) \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.3: Symbolic expression of the Coriolis matrix $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$

which we can rewrite in terms of $\dot{\mathbf{q}}$ as follows

$$P_a = \underbrace{\begin{bmatrix}\mathbf{0} & \mathbf{u}^\mathsf{T}\end{bmatrix}}_{\mathbf{Q}_a^\mathsf{T}} \underbrace{\begin{bmatrix}\mathbf{w}\\ \dot{\boldsymbol{\varphi}}\end{bmatrix}}_{\dot{\mathbf{q}}}.$$

By identifying the vectors of this equation with those of Eq. (3.3) it is clear that

$$\mathbf{Q}_a = (0,0,0,\tau_r,\tau_l,\tau_p).$$

For later developments, notice also that $\mathbf{Q}_a = \mathbf{E}\,\mathbf{u}$, where

$$\mathbf{E}\,(\mathbf{q}) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3.1.4 Generalized forces of friction

As explained in Section 1.4.2, we assume that all friction forces can be neglected except those due to viscous friction at the motor shafts. Therefore, the generalized force of friction will be

$$\mathbf{Q}_f = (0,0,0,-b_w\cdot\dot\varphi_r,-b_w\cdot\dot\varphi_l,-b_p\cdot\dot\varphi_p),$$

where b_w, and b_p are the viscous friction coefficients for the right and left wheels, and for the pivot joint, respectively. For later use, note also that

$$\mathbf{Q}_f = \mathbf{E}_f \, \dot{\mathbf{q}}$$

where

$$\mathbf{E}_f = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -b_w & 0 & 0 \\ 0 & 0 & 0 & 0 & -b_w & 0 \\ 0 & 0 & 0 & 0 & 0 & -b_p \end{bmatrix}.$$

3.2 Conventional dynamics solutions

We now wish to solve the inverse and forward dynamics problems, and obtain the dynamic model in control form $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$. To do so we will be using Lagrange's equation of motion cited in Eq. (3.1) and we will augment it with an acceleration constraint when needed. Solving for the robot model using this conventional method yields an equation for the system dynamics that requires the inversion of a 9×9 matrix. In Section 3.3 we will provide an alternative method that results in more compact expressions for the model and the inverse dynamics.

3.2.1 Inverse dynamics

We now wish to compute the inverse dynamics of our system, which consists in finding the torques $\mathbf{u} = (\tau_r, \tau_l, \tau_p)$ that correspond to a given $\ddot{\mathbf{q}}$. These torques are obtained by solving

$$\mathbf{M} \, \ddot{\mathbf{q}} + \mathbf{C} \, \dot{\mathbf{q}} + \mathbf{J}^{\mathsf{T}} \, \boldsymbol{\lambda} = \mathbf{E} \, \mathbf{u} + \mathbf{E}_f \, \dot{\mathbf{q}} \tag{3.4}$$

for $\mathbf{u}$ and $\boldsymbol{\lambda}$ (6 equations and 6 unknowns). This is the Lagragian equation adapted to our system, with the dependencies on $\mathbf{q}$ and $\dot{\mathbf{q}}$ omitted for simplicity, and with $\mathbf{Q}_a$ and $\mathbf{Q}_f$ replaced by $\mathbf{E} \, \mathbf{u}$ and $\mathbf{E}_f \, \dot{\mathbf{q}}$. To solve the problem we define $\boldsymbol{\tau}_{ID} = \mathbf{M} \, \ddot{\mathbf{q}} + (\mathbf{C} - \mathbf{E}_f) \, \dot{\mathbf{q}}$ and rewrite the previous equation as

$$\mathbf{E} \, \mathbf{u} - \mathbf{J}^{\mathsf{T}} \, \boldsymbol{\lambda} = \boldsymbol{\tau}_{ID},$$

or equivalently, as

$$\begin{bmatrix} \mathbf{E} & -\mathbf{J}^{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \boldsymbol{\tau}_{ID}.$$

It is easy to see that $[\mathbf{E} \;\; -\mathbf{J}^{\mathsf{T}}]$ is a full rank matrix irrespective of $\mathbf{q}$. This follows directly

from the expressions of $\mathbf{E}$ and $\mathbf{J}$ in the Otbot. Thus, we can write

$$\begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{E} & -\mathbf{J}^{\mathsf{T}} \end{bmatrix}^{-1} \boldsymbol{\tau}_{ID},$$

which gives the solution to the inverse dynamics:

$$\mathbf{u} = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{E} & -\mathbf{J}^{\mathsf{T}} \end{bmatrix}^{-1} \boldsymbol{\tau}_{ID}. \tag{3.5}$$

3.2.2 Forward dynamics

The forward dynamics problem consists in finding the acceleration $\ddot{\mathbf{q}}$ that corresponds to a given action $\mathbf{u}$. Similarly to the inverse problem, we need to solve Eq. (3.4), but this time for $\ddot{\mathbf{q}}$, which gives us a system with 6 equations in 9 unknowns (6 coordinates in $\ddot{\mathbf{q}}$ and 3 in $\boldsymbol{\lambda}$). Thus, we need three more equations, which can be obtained by taking the time derivative of $\mathbf{J}\ \dot{\mathbf{q}} = \mathbf{0}$, which gives

$$\mathbf{J}\ \ddot{\mathbf{q}} + \dot{\mathbf{J}}\ \dot{\mathbf{q}} = \mathbf{0},$$

or equivalently

$$\mathbf{J}\ \ddot{\mathbf{q}} = -\dot{\mathbf{J}}\ \dot{\mathbf{q}}.$$

The latter equation is called the acceleration constraint of the robot. By combining it with Eq. (3.4), we obtain a solvable linear system of 9 equations with 9 unknowns with the form:

$$\begin{bmatrix} \mathbf{M} & \mathbf{J}^{\mathsf{T}} \\ \mathbf{J} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{E}\,\mathbf{u} + (\mathbf{E}_f - \mathbf{C})\ \dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\ \dot{\mathbf{q}} \end{bmatrix}.$$

Since $\mathbf{M}$ is positive-definite and $\mathbf{J}$ is full row rank, the matrix on the left-hand side can be inverted to write

$$\begin{bmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}^{\mathsf{T}} \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} + (\mathbf{E}_f - \mathbf{C})\ \dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\ \dot{\mathbf{q}} \end{bmatrix},$$

which gives us the solution for the forward dynamics:

$$\ddot{\mathbf{q}} = \begin{bmatrix} \mathbf{I}_6 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{M} & \mathbf{J}^{\mathsf{T}} \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} + (\mathbf{E}_f - \mathbf{C})\ \dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\ \dot{\mathbf{q}} \end{bmatrix}. \tag{3.6}$$

3.3 Cancelling the multipliers

Eqs. (3.5) and (3.6) require the inversion of a 6×6 and a 9×9 matrix respectively, which may be costly operations when, as in this project, we have to repeat them many times. We next see that, by exploiting two parameterizations of the feasible velocities $\dot{\mathbf{q}}$, we can find simpler solutions that require, in each case, no matrix inverse, and the inverse of a 6×6 matrix only. These parameterizations are

$$\begin{bmatrix} \dot{\mathbf{p}} \\ \dot{\boldsymbol{\varphi}} \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{I}_3 \\ \mathbf{M}_{IIK} \end{bmatrix}}_{\boldsymbol{\Lambda}} \dot{\mathbf{p}} \quad \rightarrow \quad \dot{\mathbf{q}} = \boldsymbol{\Lambda} \cdot \dot{\mathbf{p}}, \tag{3.7}$$

$$\begin{bmatrix} \dot{\mathbf{p}} \\ \dot{\boldsymbol{\varphi}} \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{M}_{FIK} \\ \mathbf{I}_3 \end{bmatrix}}_{\boldsymbol{\Delta}} \dot{\boldsymbol{\varphi}} \quad \rightarrow \quad \dot{\mathbf{q}} = \boldsymbol{\Delta} \cdot \dot{\boldsymbol{\varphi}}, \tag{3.8}$$

where $\mathbf{M}_{FIK}$ and $\mathbf{M}_{IIK}$ are the matrices used to solve the forward and inverse instantaneous kinematic problems (Section 2.3).

3.3.1 Inverse dynamics

Again, we wish to solve Eq. (3.5) for $\mathbf{u}$. To do this, we can multiply it by $\boldsymbol{\Delta}^\mathsf{T}$ to obtain

$$\boldsymbol{\Delta}^\mathsf{T}\mathbf{M}\,\ddot{\mathbf{q}} + \boldsymbol{\Delta}^\mathsf{T}\mathbf{C}\,\dot{\mathbf{q}} + \boldsymbol{\Delta}^\mathsf{T}\mathbf{J}^\mathsf{T}\boldsymbol{\lambda} = \boldsymbol{\Delta}^\mathsf{T}\mathbf{E}\,\mathbf{u} + \boldsymbol{\Delta}^\mathsf{T}\mathbf{E}_f\,\dot{\mathbf{q}}, \tag{3.9}$$

which introduces two simplifications. On the one hand, it is not difficult to see that

$$\boldsymbol{\Delta}^\mathsf{T}\mathbf{J}^\mathsf{T} = \mathbf{0}. \tag{3.10}$$

This can be explained using the constraint in Eq. (2.2), which is satisfied only if $\dot{\mathbf{q}}$ is admissible. Since Eq. (3.8) also provides admissible $\dot{\mathbf{q}}$ for any $\dot{\boldsymbol{\varphi}}$, we have

$$\mathbf{J}\,\dot{\mathbf{q}} = \mathbf{J}\boldsymbol{\Delta}\,\dot{\boldsymbol{\varphi}} = \mathbf{0}.$$

As the earlier result must hold for any $\dot{\boldsymbol{\varphi}}$, it must be $\mathbf{J}\boldsymbol{\Delta} = \mathbf{0}$ and by taking the transpose we find that Eq. (3.10) must be true. On the other hand, we can directly simplify the right-hand side of Eq. (3.9) since we have

$$\boldsymbol{\Delta}^\mathsf{T}\mathbf{E} = \begin{bmatrix} \mathbf{M}_{FIK} & \boldsymbol{I}_3 \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ \mathbf{I}_3 \end{bmatrix} = \mathbf{I}_3. \tag{3.11}$$

Therefore, by substituting Eqs. (3.10) and (3.11) in Eq. 3.9 we obtain a closed-form expression for the inverse dynamics in which no matrix inverse is involved:

$$\mathbf{u} = \boldsymbol{\Delta}^{\mathsf{T}}\mathbf{M}\,\ddot{\mathbf{q}} + \boldsymbol{\Delta}^{\mathsf{T}}\left(\mathbf{C} - \mathbf{E}_f\right)\,\dot{\mathbf{q}}. \tag{3.12}$$

3.3.2 Forward dynamics

We now wish to find a simplified version of the forward dynamics. To do this we will first compute the dynamic model in task-space coordinates, which provides the time evolution of the $\mathbf{p}$ state variables. This can be done by using Eq. (3.7) and its derivative

$$\ddot{\mathbf{q}} = \boldsymbol{\Lambda}\,\ddot{\mathbf{p}} + \dot{\boldsymbol{\Lambda}}\,\dot{\mathbf{p}}. \tag{3.13}$$

By substituting them into Eq. (3.12), we find:

$$\begin{aligned}\boldsymbol{\Delta}^{\mathsf{T}}\mathbf{M}(\boldsymbol{\Lambda}\,\ddot{\mathbf{p}} + \dot{\boldsymbol{\Lambda}}\,\dot{\mathbf{p}}) + \boldsymbol{\Delta}^{\mathsf{T}}(\mathbf{C} - \mathbf{E}_f)\boldsymbol{\Lambda}\,\dot{\mathbf{p}} = \mathbf{u},\\ \boldsymbol{\Delta}^{\mathsf{T}}\mathbf{M}\boldsymbol{\Lambda}\,\ddot{\mathbf{p}} + \boldsymbol{\Delta}^{\mathsf{T}}\mathbf{M}\dot{\boldsymbol{\Lambda}}\,\dot{\mathbf{p}} + \boldsymbol{\Delta}^{\mathsf{T}}(\mathbf{C} - \mathbf{E}_f)\boldsymbol{\Lambda}\,\dot{\mathbf{p}} = \mathbf{u},\end{aligned}$$

finally giving us

$$\underbrace{\boldsymbol{\Delta}^{\mathsf{T}}\mathbf{M}\boldsymbol{\Lambda}}_{\bar{\mathbf{M}}}\,\ddot{\mathbf{p}} + \underbrace{\boldsymbol{\Delta}^{\mathsf{T}}\left[\mathbf{M}\dot{\boldsymbol{\Lambda}} + (\mathbf{C} - \mathbf{E}_f)\boldsymbol{\Lambda}\right]}_{\bar{\mathbf{C}}}\,\dot{\mathbf{p}} = \mathbf{u},$$

where $\bar{\mathbf{M}}$ is non singular because $\boldsymbol{\Delta}^{\mathsf{T}}$, $\mathbf{M}$ and $\boldsymbol{\Lambda}$ are all full rank. We now have an expression for the dynamic model in task-space coordinates:

$$\bar{\mathbf{M}}\ddot{\mathbf{p}} + \bar{\mathbf{C}}\dot{\mathbf{p}} = \mathbf{u}. \tag{3.14}$$

The earlier expression does not allow us to solve for $\ddot{\mathbf{q}}$ yet, but note that Eq. (3.13) can be expanded into:

$$\begin{bmatrix}\ddot{\mathbf{p}}\\ \ddot{\boldsymbol{\varphi}}\end{bmatrix} = \begin{bmatrix}\mathbf{I}_3\\ \mathbf{M}_{IIK}\end{bmatrix}\ddot{\mathbf{p}} + \begin{bmatrix}\mathbf{0}\\ \dot{\mathbf{M}}_{IIK}\end{bmatrix}\dot{\mathbf{p}},$$

whose second row gives us an expression of $\ddot{\boldsymbol{\varphi}}$ in terms of $\ddot{\mathbf{p}}$ and $\dot{\mathbf{p}}$. By combining this expression with Eq. (3.14) and rearranging the terms we obtain the system:

$$\begin{cases}\bar{\mathbf{M}}\,\ddot{\mathbf{p}} = \mathbf{u} - \bar{\mathbf{C}}\,\dot{\mathbf{p}}\\ -\mathbf{M}_{IIK}\,\ddot{\mathbf{p}} + \ddot{\boldsymbol{\varphi}} = \dot{\mathbf{M}}_{IIK}\,\dot{\mathbf{p}}\end{cases}$$

or in block form

$$\underbrace{\begin{bmatrix} \bar{\mathbf{M}} & \mathbf{0} \\ -\mathbf{M}_{IIK} & \mathbf{I}_3 \end{bmatrix}}_{\mathbf{K}(\mathbf{q})} \underbrace{\begin{bmatrix} \ddot{\mathbf{p}} \\ \ddot{\boldsymbol{\varphi}} \end{bmatrix}}_{\ddot{\mathbf{q}}} = \underbrace{\begin{bmatrix} \mathbf{u} - \bar{\mathbf{C}}\,\dot{\mathbf{p}} \\ \dot{\mathbf{M}}_{IIK}\,\dot{\mathbf{p}} \end{bmatrix}}_{\mathbf{b}_{FD}(\mathbf{q},\dot{\mathbf{q}},\mathbf{u})},$$

which easily gives us the forward dynamics by inverting of the 6×6 matrix on the left-hand side:

$$\ddot{\mathbf{q}} = \mathbf{K}^{-1}\,\mathbf{b}_{FD}. \tag{3.15}$$

It is now simple to write the dynamic model which expresses the derivative of the state variables $\dot{\mathbf{x}}$ as a function of the state variables $\mathbf{x}$ and the control action $\mathbf{u}$. To do this, we only need to augment Eq. (3.15) with the trivial equation $\dot{\mathbf{q}} = \dot{\mathbf{q}}$ to obtain

$$\begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}} \\ \mathbf{K}^{-1}\,\mathbf{b}_{FD} \end{bmatrix}, \tag{3.16}$$

which gives the robot model in the usual form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}).$$

4

Parameter identification

In this chapter we develop a method to identify the dynamical parameters of the Otbot. To this end we resort to nonlinear grey-box model identification, which searches for the set of parameters that yield the best fit between sensor signals recorded by the robot, and the values of such signals as predicted by the model. As we will see, this will entail finding the minimum of a nonlinear function of the sought parameters. The chapter starts with a short introduction to system identification by prediction-error minimization, then explains the identification process we propose, and finally describes a number of tests we have done to validate our approach in simulation. Our results show that the approach is robust to the presence of noise in the signals, and that it converges fairly well in a sufficiently-large region around the actual parameter values.

4.1 Prediction error minimization

Suppose that the ordinary differential equation that describes the dynamic model of our robot is given by

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, \mathbf{u}, \boldsymbol{p}), \tag{4.1}$$

where $\mathbf{x}$ is the robot state, $\mathbf{u}$ are the control actions, and $\boldsymbol{p}$ is the vector of system parameters. Typically, we do not have sensors to measure each one of the variables in $\mathbf{x}$. Instead, the robot sensors provide a vector $\mathbf{y}$ of outputs whose functional dependence on $\mathbf{x}$, $\mathbf{u}$, and $\boldsymbol{p}$,

$$\mathbf{y} = \mathbf{G}(\mathbf{x}, \mathbf{u}, \boldsymbol{p}), \tag{4.2}$$

is known in advance. In this situation we can estimate the value of $\boldsymbol{p}$ as follows. Assuming the initial state $\mathbf{x}_0 = \mathbf{x}(0)$ is known, we command the robot under some control function $\mathbf{u}(t)$ for $t = 0$ to t_f, while recording the values of $\mathbf{u}(t)$ and $\mathbf{y}(t)$ simultaneously. Since $\mathbf{u}(t)$ and $\mathbf{y}(t)$ are

usually managed with digital means, they have the form of sampled-data sequences

$$\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_N,$$
$$\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_N,$$

whose values correspond to the time instants $t_0 < t_1 < \ldots < t_N$. We can then construct the fitting function

$$\varepsilon(\boldsymbol{p}) = \sum_{k=0}^{N} \left\| \mathbf{y}_k - \mathbf{y}_k^{\text{pred}} \right\|^2, \tag{4.3}$$

where

$$\mathbf{y}_k^{\text{pred}} = \mathbf{G}(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{p}), \tag{4.4}$$

and try to find the values of $\boldsymbol{p}$ that minimize $\varepsilon(\boldsymbol{p})$. At a first glance, it looks like the $\mathbf{x}_k$ values are unknown, but we can obtain them by solving a sequence of initial value problems with an appropriate numerical integration method to express $\mathbf{x}_k$ as a function of $\mathbf{x}_0$, $\mathbf{u}_0, \ldots, \mathbf{u}_{k-1}$, and $\boldsymbol{p}$. Thus, $\varepsilon(\boldsymbol{p})$ depends ultimately on $\boldsymbol{p}$, as $\mathbf{x}_0$, $\mathbf{u}_0, \ldots, \mathbf{u}_{N-1}$, $\mathbf{y}_0, \ldots, \mathbf{y}_N$ are all known a priori. In the literature, $\varepsilon(\boldsymbol{p})$ is called a prediction error (or loss) function [8] as it provides a measure of how well the parameters $\boldsymbol{p}$ explain the evolution of the outputs under the model assumed.

To minimize $\varepsilon(\boldsymbol{p})$ we can proceed in a number of ways, but we will here use the Trust Region Reflective Algorithm [9, 10], which is a common approach in system identification [11]. The idea of this algorithm is to iteratively replace $\varepsilon(\boldsymbol{p})$ by a quadratic function $\mathbf{Q}(\boldsymbol{p})$ that approximates $\varepsilon(\boldsymbol{p})$ in a neighborhood $\mathcal{N}$ around the current point $\boldsymbol{p} = \boldsymbol{p}_{curr}$. This neighborhood is the mentioned trust region. At each iteration, an update of $\boldsymbol{p}_{curr}$ is attempted by finding the point $\boldsymbol{p}_{new}$ that minimizes $\mathbf{Q}(\boldsymbol{p})$ subject to $\boldsymbol{p} \in \mathcal{N}$. If

$$\varepsilon(\boldsymbol{p}_{new}) < \varepsilon(\boldsymbol{p}_{curr}),$$

we set $\boldsymbol{p}_{curr} = \boldsymbol{p}_{new}$ and iterate the process again. Otherwise, $\boldsymbol{p}_{curr}$ remains unchanged, $\mathcal{N}$ is shrunk, and a new trial step is attempted.

The function $\mathbf{Q}(\boldsymbol{p})$ is typically the second-order Taylor expansion of $\varepsilon(\boldsymbol{p})$ around $\boldsymbol{p}_{curr}$,

$$\mathbf{Q}(\boldsymbol{p}) = \varepsilon(\boldsymbol{p}_{curr}) + \mathbf{g}^{\mathsf{T}} \cdot (\boldsymbol{p} - \boldsymbol{p}_{curr}) + \frac{1}{2}(\boldsymbol{p} - \boldsymbol{p}_{curr})^{\mathsf{T}} \cdot \mathbf{H} \cdot (\boldsymbol{p} - \boldsymbol{p}_{curr}),$$

where $\mathbf{g}$ and $\mathbf{H}$ are the gradient and Hessian of $\varepsilon(\boldsymbol{p})$ evaluated at $\boldsymbol{p}_{curr}$. Since $\varepsilon(\boldsymbol{p})$ typically has a complex expression, $\mathbf{g}$ and $\mathbf{H}$ are obtained by finite differences, so we only need to evaluate $\varepsilon(\boldsymbol{p})$ repeatedly to obtain such derivatives. This is done by computing the $\mathbf{x}_k$ values from $\boldsymbol{p}$, $\mathbf{x}_0$,

$\mathbf{u}_0, \ldots, \mathbf{u}_{k-1}$ using some integration method, then evaluating the outputs $\mathbf{y}_k^{\text{pred}}$ using Eq. (4.4), and finally computing the sum in Eq. (4.3).

In this project, we have used the MATLAB implementation of the earlier algorithm, which is easily set up and run using the *nlgreyest* method from the System Identification Toolbox [12]. To obtain the $\mathbf{x}_k$ values in Eq. (4.4) we have used the *ode45* method, which implements the Dormand-Prince algorithm of order 4/5 [13]. This technique uses a Runge-Kutta method of fourth-order for time stepping, and a fifth-order method for error estimation and step adjustment. The obtained simulations are very precise therefore.

To apply the integration method, one has to decide how $\mathbf{u}(t)$ is to be interpolated from $\mathbf{u}_0, \ldots, \mathbf{u}_N$. In this project we assume that the real robot is controlled with a zero-order-hold filter, so we also use this filter to interpolate $\mathbf{u}(t)$ in simulation. This means that $\mathbf{u}(t) = \mathbf{u}_k$ for $t_k \leq t < t_{k+1}$, $k = 0, \ldots, N-1$.

4.2 Parameter identification sequence

In order to use the inverse and forward dynamics equations (3.4) and (3.6) in our simulations, we need to determine the parameters that appear in the matrices $\mathbf{M}$, $\mathbf{C}$, $\mathbf{J}$ and $\mathbf{E}_f$. All parameters are assumed to be constant except those of the platform: its mass m_p, moment of inertia I_p, and coordinates of the c.o.m. (x_F, y_F), which may be different from one task to another depending on the load carried by the robot. For convenience we denote the values of the unloaded platform parameters as $m_{p,0}$, $I_{p,0}$, $x_{F,0}$ and $y_{F,0}$, respectively, and we will refer to the varying parameters m_p, I_p, x_F, y_F, as the working platform parameters.

Table 4.1 summarizes all the parameters with their nominal values. We assume that some

Symbol	Meaning	Value	Unit
l_1	Pivot offset relative to the wheels axis	0.25	m
l_2	One half of the wheels separation	0.20	m
r	Wheel radius	0.10	m
(x_B, y_B)	Coords. of c.o.m. B of the chassis in chassis frame	$(-0.13, 0)$	m
$(x_{F,0}, y_{F,0})$	Coords. of c.o.m. F of the unloaded platform in platform frame	$(0, 0)$	m
m_c	Mass of the chassis including the wheels	109.14	kg
$m_{p,0}$	Mass of the platform without load	21.95	kg
I_c	Vertical moment of inertia of the chassis plus wheels at B	1.30	kg m^2
$I_{p,0}$	Vertical moment of inertia of the unloaded platform at F	2.22	kg m^2
I_a	Axial moment of inertia of a wheel	1.04×10^{-2}	kg m^2
b_w	Viscous friction coefficient at the wheel's shaft	0.18	kg m^2 s^{-1}
b_p	Viscous friction coefficient at the pivot shaft	0.24	kg m^2 s^{-1}

Table 4.1: Nominal parameters of the robot assumed in system identification tests.

Step 1 Basic parameters	Step 2 Chassis parameters	Step 3 Working-platform parameters
$I_{p,0}$	m_c	m_p
I_a	I_c	I_p
b_w	x_B	x_F
b_p	y_B	y_F

Table 4.2: The three identification steps.

of them are already known or that can be directly measured, so we will not specify any identification procedure for them. This applies to the parameters l_1, l_2, r, $m_{p,0}$, $x_{F,0}$, and $y_{F,0}$, whose nominal values will be taken for granted.

The identification process is divided into three consecutive steps (Table 4.2), as described next:

1 **Identification of basic parameters:** This step involves parameters of individual parts of the robot that can be identified by the activation of a single motor. This includes the vertical moment of inertia of the unloaded platform, $I_{p,0}$, the moment of inertia of a wheel with respect to its rotation axis, I_a, and their corresponding friction coefficients, b_p, b_w. We assume the center of mass of the platform is at the pivot point, and that both wheels are identical and have the same friction coefficient b_w.

2 **Identification of the chassis parameters:** In his step, we try to determine the four parameters of the chassis: m_c, I_c, x_B, y_B. For this, we assume that the platform is unloaded, so that its parameters are all previously known or obtained from step 1.

3 **Identification of the working platform parameters:** In this step we will identify the parameters corresponding to the platform together with the unknown load placed on it: m_p, I_p, x_F, y_F, which may be different for each experiment. In this case we make use of the parameter values already identified in steps 1 and 2. This step must be repeated each time the load carried by the robot changes.

4.3 Identification results

The identification process requires the determination of the $\mathbf{u}_k$ and $\mathbf{y}_k$ sequences defining the fitting function (4.3). In a real robot, this would involve executing some control action sequence $\mathbf{u}_k$ in the real world while recording the outputs $\mathbf{y}_k$ provided by the sensor readings. Since, in our case, the physical robot is not available, we will obtain the outputs $\mathbf{y}_k$ by an accurate

Parameter	Nominal value	Initial guess	Estimated value	Absolute error	Unit
b_w	0.18	0.09	0.18	8.61×10^{-6}	$\mathrm{kg\ m^2\ s^{-1}}$
I_a	0.01	0.01	0.01	7.40×10^{-6}	$\mathrm{kg\ m^2}$
b_p	0.24	0.12	0.24	4.90×10^{-4}	$\mathrm{kg\ m^2\ s^{-1}}$
$I_{p,0}$	2.22	1.11	2.22	2.77×10^{-4}	$\mathrm{kg\ m^2}$

Table 4.3: Results of the basic parameters identification.

simulation of the dynamics using the nominal parameter values. To be more realistic, we will add some Gaussian noise to the output of each sensor, with standard deviations that we consider plausible for each kind of sensor. Then, in practice, for each identification experiment, we perform multiple simulations with the same control action sequence: a first one that plays the role of the real robot using the nominal parameters to obtain the corresponding $\mathbf{y}_k$ sequence, and one more for each set of parameters to test to obtain the $\mathbf{y}_k^{\mathrm{pred}}$ sequence, all simulations using the same dynamic equations and integration algorithm.

4.3.1 Identification of basic parameters

The parameters to be identified in this step are the moment of inertia I and friction coefficient b of two robot parts: the unloaded platform and a wheel. In both cases there is a motor acting directly on the corresponding axis, and we assume that both motors are equipped with angular encoders able to provide an accurate measure of their angular velocity. The identification procedure is analogous in both cases. It consists in applying a constant torque to the corresponding axis and registering the readings of the encoder. Thus, in this case, the output function in Eq. (4.2) simply provides the angular velocity. The dynamic equation governing the motion is

$$u - b \cdot \dot{\varphi} = I \cdot \ddot{\varphi} \tag{4.5}$$

where $(u, b, \varphi, I) = (\tau_p, b_p, \varphi_p, I_{p,0})$ in the case of the platform, and $(\tau_r, b_w, \varphi_r, I_a)$ in the case of the (right) wheel. In each experiment, two parameters must be identified simultaneously, I and b.

To perform the experiments, we applied a constant torque of $6\mathrm{Nm}$ during $1.5\mathrm{s}$ for the platform and $0.5\mathrm{s}$ for the wheel, sampling the angular velocity at a frequency of $100\mathrm{Hz}$. The simulated encoder readings are corrupted by a Gaussian noise with a $\sigma = 0.01\mathrm{rad/s}$ standard deviation, which reflects the good precision usually provided by digital encoders. The initial guess (the initial $\boldsymbol{p}_{curr}$ of the Trust Region Reflective Algorithm) and the final estimated value for each parameter are shown in Table 4.3. The results of the experiments are shown in Fig. 4.1.

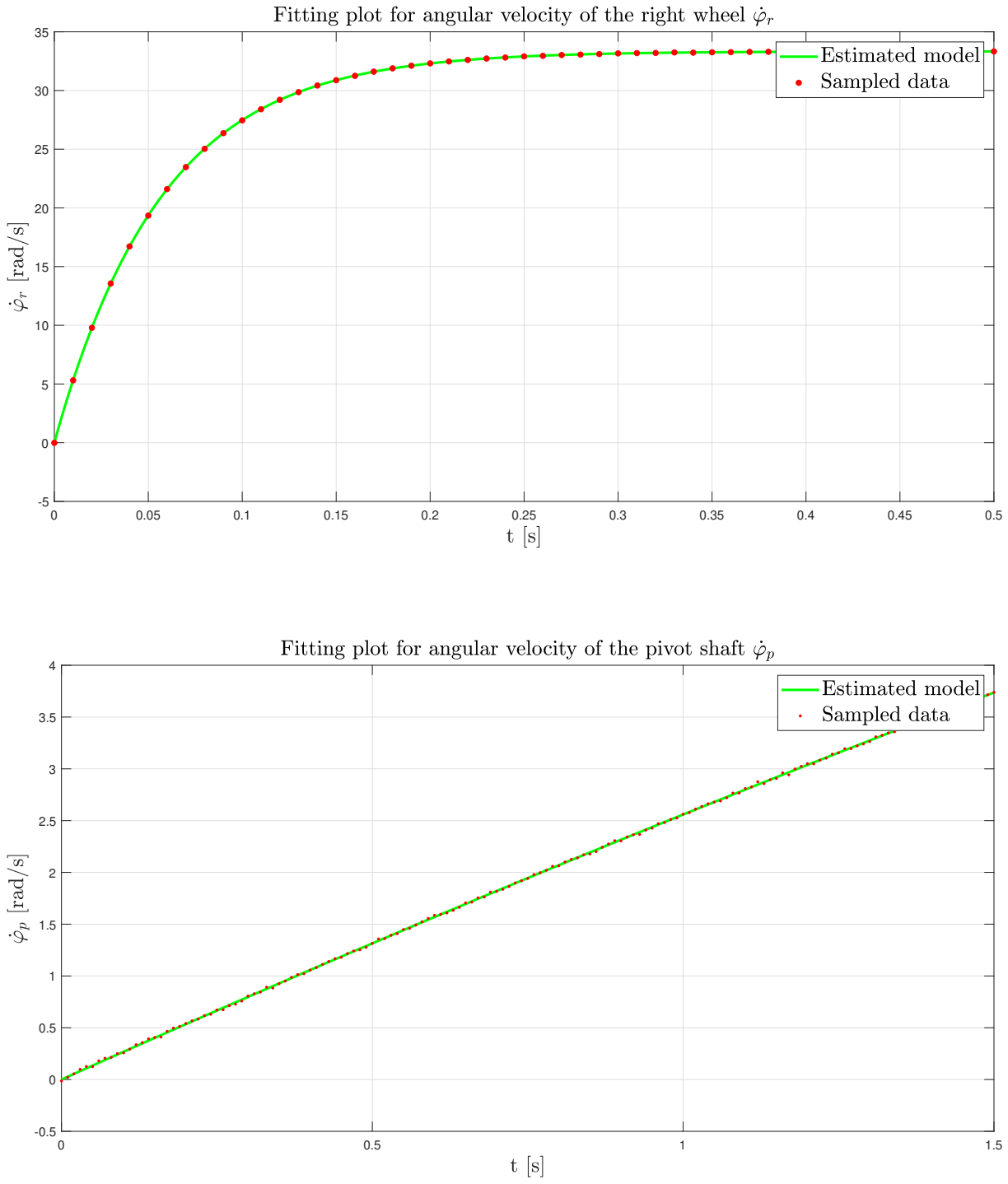


Figure 4.1: Fitting plots of the experiments for the wheel and platform parameter identification.

Parameter	Nominal value	Initial guess	Estimated value	Absolute error	Unit
m_c	109.14	54.57	109.12	0.02	kg
I_c	1.30	0.65	1.31	8.87×10^{-4}	kg m^2
x_B	-0.13	-0.07	-0.13	1.72×10^{-5}	m
y_B	0.00	0.25	-4.31×10^{-5}	4.31×10^{-5}	m

Table 4.4: Results of the chassis parameters identification.

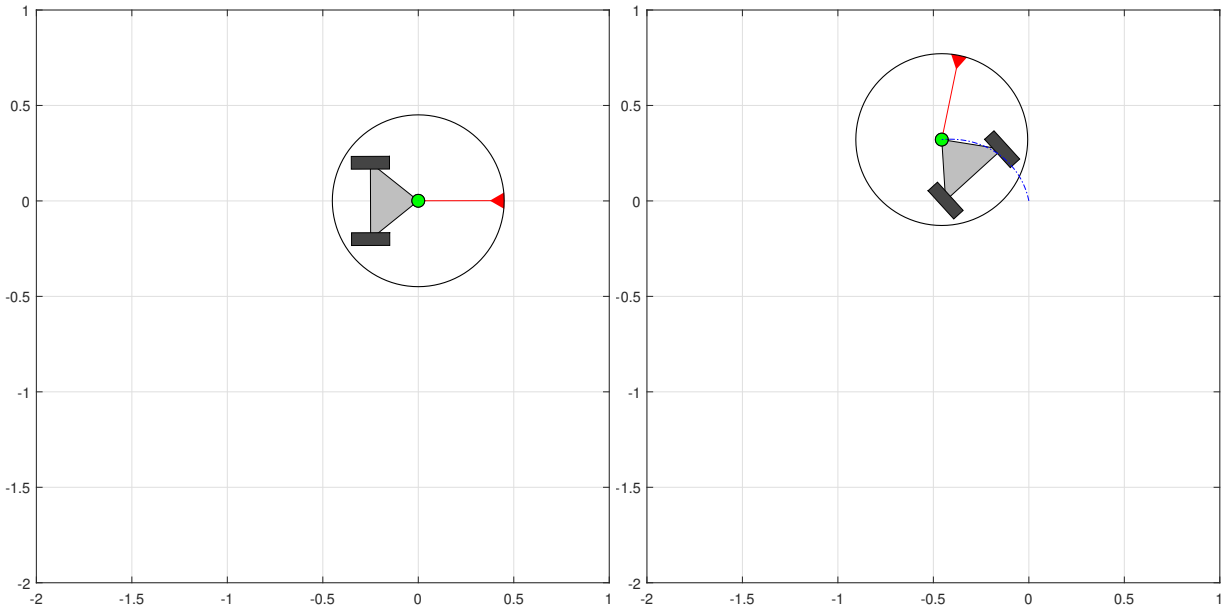
4.3.2 Identification of the chassis parameters

In this step, the four parameters of the chassis must be identified simultaneously by driving the robot with some control action sequence. To obtain a precise identification of all four parameters, it is necessary that the trajectory followed in the experiment provides sufficient information to reveal the influence of each parameter in the trajectory, otherwise, some parameter could become irrelevant, making it unidentifiable, or maybe the observed outputs could be obtained with different combinations of parameters. Several actuation patterns involving the three motors have been tried, using equal or different torque profiles, combining them in different ways, putting them in and out of synchrony, and so on. Our purpose was to find an action sequence able to provide the required information in a short time and using limited torques, so that the robot displacement is confined in a reduced area. The final choice that produced good results was to apply constant torques to each motor during 3s with the following values: $\tau_r = 6\text{Nm}$, $\tau_l = -10\text{Nm}$ and $\tau_p = 6\text{Nm}$. The resulting trajectory is shown in Fig. 4.2.

The output signals $\mathbf{y}_k$ are obtained assuming that the robot is equipped with an IMU attached to the platform that provides the instantaneous values of the acceleration of the pivot point $(\ddot{x}, \ddot{y})$ in basis B'', and the angular velocity of the platform $\dot{\alpha}$. As in step 1, we sampled the sensor at a frequency of 100Hz. The standard deviation of the noise applied to the sensor readings has been computed according to the usual formula for an IMU

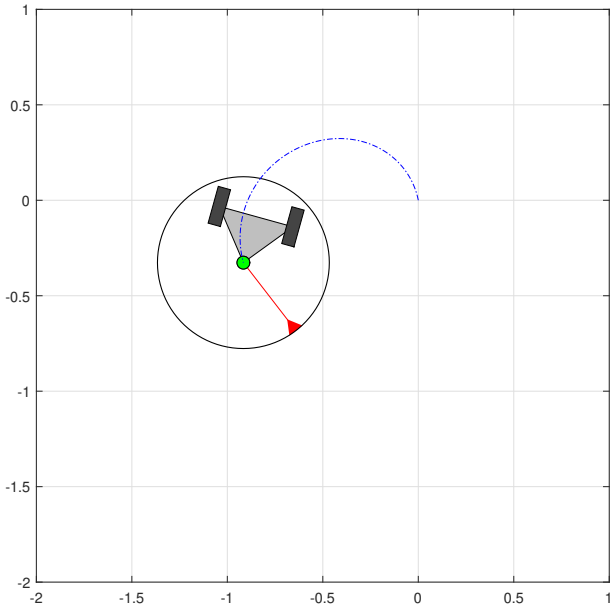
$$\sigma_{IMU} = ND\sqrt{SR}, \tag{4.6}$$

where ND is the noise density, a parameter provided by the manufacturer that we assume to have a value of $1.37 \times 10^{-3}\text{m s}^{-2}\ \text{Hz}^{-1/2}$, and SR is the sample rate, that in our case is 100Hz. By replacing these values into Eq. (4.6) we end up with a standard deviation of $\sigma_{IMU} = 13.73 \times 10^{-3}$. The initial guess and the final estimated value for each parameter are shown in Table 4.4. The result of the experiment is shown in Fig. 4.3.

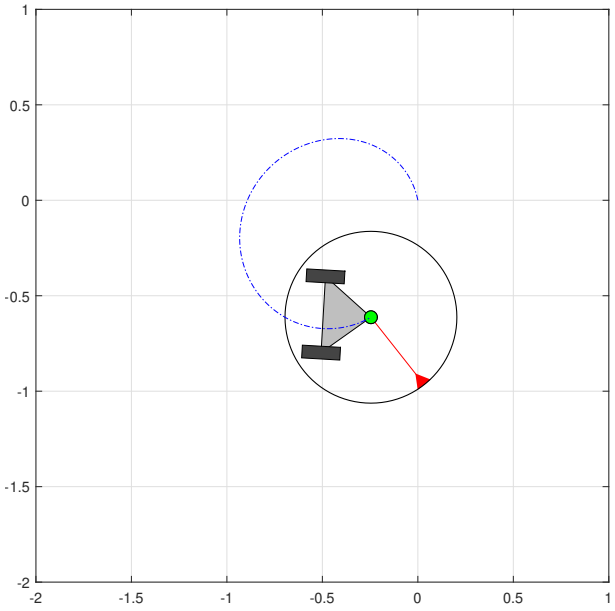


(a) Otbot at the initial configuration.

(b) Otbot configuration after 1s.



(c) Otbot configuration after 2s.



(d) Otbot configuration after 3s.

Figure 4.2: Experiment for the identification of the chassis parameters. The path followed by the pivot joint is shown in blue.

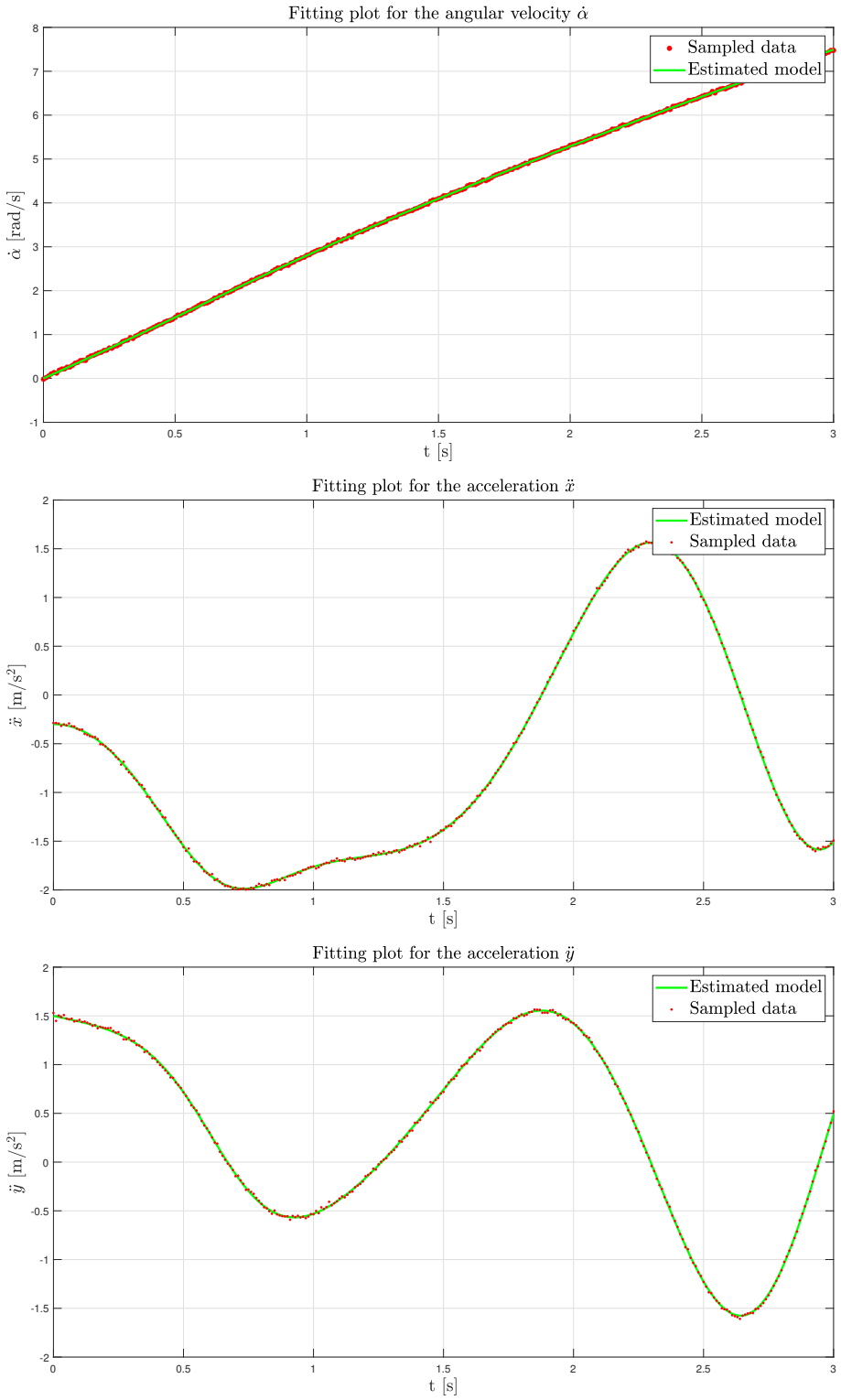


Figure 4.3: Fitting plots of the experiment for the chassis parameters identification.

4.3.3 Identification of the working platform parameters

This step is conceptually very similar to the previous one, since the four parameters we have to identify for the working platform correspond to the four parameters identified in step 1 for the chassis. The difference is that, instead of fixing the platform parameters and estimating those of the chassis, we will fix the parameters of the chassis and try to estimate those of the platform, which now we assume to be unknown. A further difference with step 1 is the larger deviation from the nominal parameter values imposed to the initial guess. This is required because we assume that the robot should be able to carry loads several times heavier than the platform itself, so that we may have only a very rough idea of its possible mass, moment of inertia and position of the c.o.m., and it is important to check that the identification process is reliable in the wide range of working conditions that can be expected.

In order to assess the robustness of the identification in front of the deviation of the initial guess, we performed a series of tests with increasing amounts of deviation. For convenience reasons, instead of using different weights and positions of the load and perform the identification with the same initial guess, we proceed in the opposite way, i.e., we increase the deviation of the initial guess while keeping the platform parameters unmodified. This allows us to use the same experiment as in step 1, with the same simulated sensor data for all tests, thus avoiding the need to simulate a new trajectory for each load configuration. This strategy permits to test many instances of parameter deviations of different magnitudes in a fast way.

The range of variations of the initial guess to test has been determined from the assumption that the mass of the load can reach a maximum of $500\mathrm{kg}$, and its position on the platform may be arbitrary within a circle with a radius of $0.45\mathrm{m}$ around the pivot. To avoid a proliferation of tests, we build each initial guess by altering the load mass and distance to the pivot with the same percentage of their maximum allowed values, which is a rather unfavorable choice. For the moment of inertia, we take a value compatible with the selected mass and distance according to the Steiner theorem. Figure 4.4 shows the sensitivity plots relating the absolute error on each parameter as a function of the amount of deviation. It can be seen that except for a sporadic outlier, all parameters are identified with great precision when the amount of deviation is as large as more than 25% of the maximum allowed. It has to be noted that, despite the trajectory of the experiment has a duration of $3\mathrm{s}$, good results are already obtained after the first second, and thus, all the tests have been made using only $1\mathrm{s}$ of simulation.

To illustrate the accuracy of the estimation, Table 4.5 shows the results obtained for a 25% deviation corresponding to a mass of the load of $120\mathrm{kg}$ (which corresponds to a parameter value $m_p = 146.95\mathrm{kg}$ when added to that of the unloaded platform), and a position of its center of mass $(x_F, y_F) = (0.11, 0.11)\mathrm{m}$.

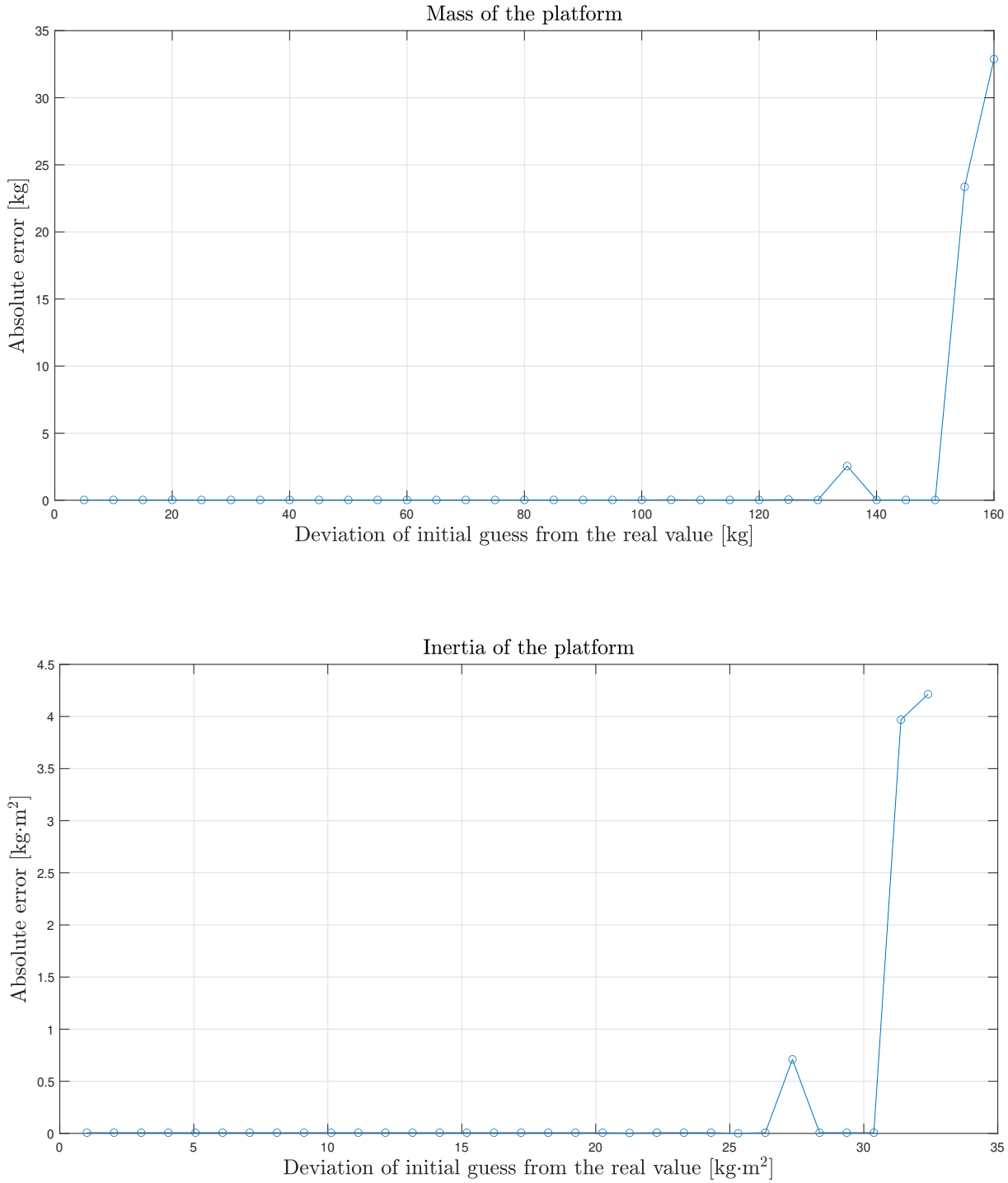


Figure 4.4: Sensitivity plots for the inertial parameters of the working platform. The curves show the absolute error in the mass and moment of inertia of the platform as a function of the deviation of their initial guesses from the real values.

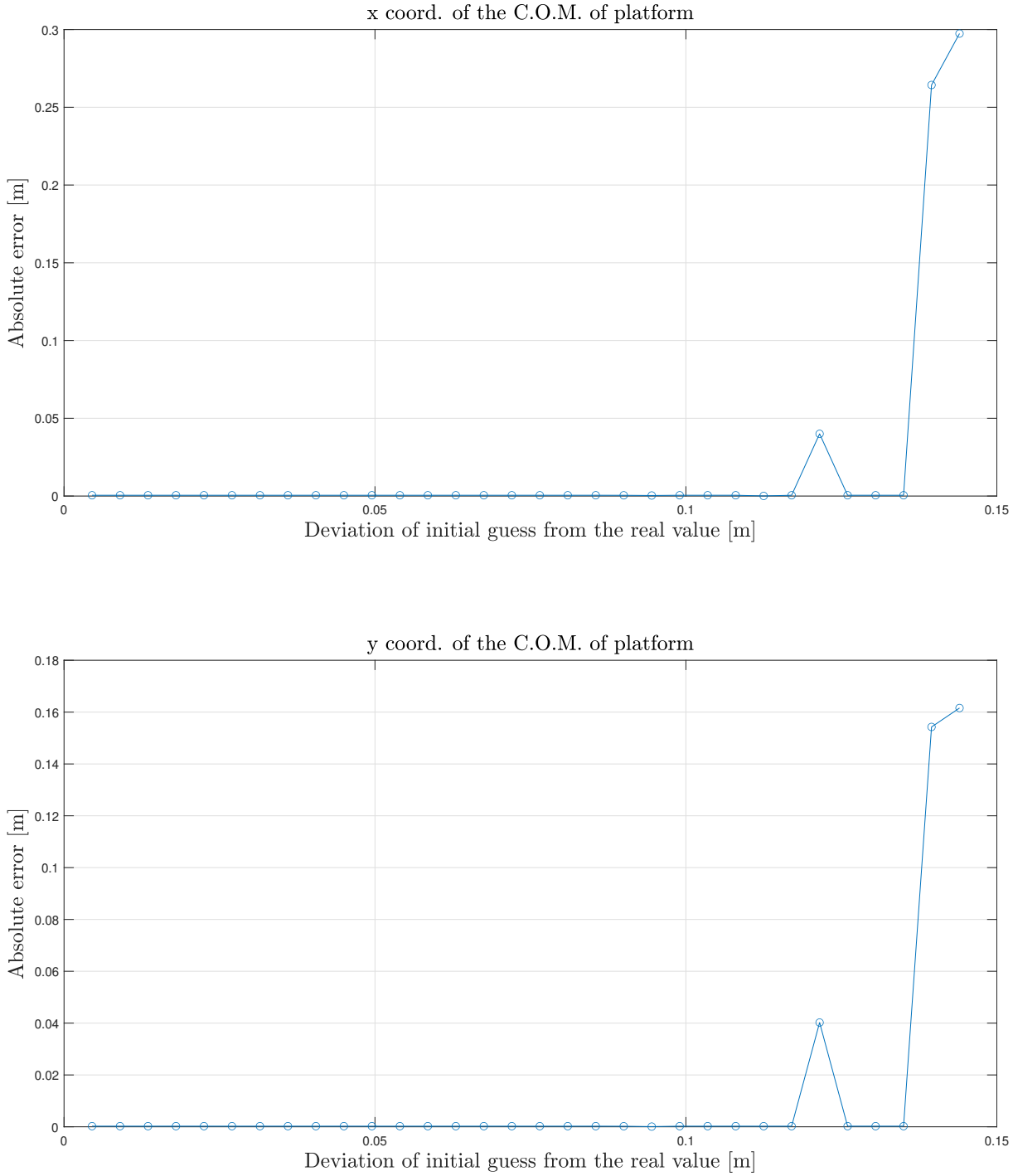


Figure 4.5: Sensitivity plots for the coordinates of the center of mass of the working platform. The curves show the absolute error in the coordinates of the center of mass of the platform as a function of the deviation of their initial guesses from the real values.

Parameter	Nominal value	Initial guess	Estimated value	Absolute error	Unit
m_p	21.95	146.95	21.99	0.05	kg
I_p	2.22	5.94	2.22	1.15×10^{-3}	kg m^2
x_F	0.00	0.11	7.50×10^{-5}	7.50×10^{-5}	m
y_F	0.00	0.11	-2.32×10^{-4}	2.32×10^{-4}	m

Table 4.5: Results of the working platform parameters identification.

5

Tracking control

With the model in Eq. (3.16) we can now move on to designing a controller to make the robot follow a desired trajectory. To do so, we have designed a globally stable computed-torque controller capable of tracking any desired trajectory from any starting state. This controller will also allow the robot to compensate unforeseen trajectory deviations or force perturbations.

5.1 A computed-torque controller

This type of controller can only stabilize as many variables as the number of action variables. Since there are only three control variables in the Otbot (two wheels and one pivot), we need to base this controller on an equation of motion that uses only three state variables. Fortunately, we have already computed the equation of motion in task-space coordinates $\mathbf{p} = (x, y, \alpha)$ in Eq. (3.14), which we here recall:

$$\bar{\mathbf{M}}\,\ddot{\mathbf{p}} + \bar{\mathbf{C}}\,\dot{\mathbf{p}} = \mathbf{u}. \tag{5.1}$$

This equation is ideal since it will let us control the $\mathbf{p}$-state variables, which are those typically used to specify a task or mission for the robot. Thus, our goal is to find a control law that makes the robot trajectory $\mathbf{p}(t)$ asymptotically convergent to a desired trajectory $\mathbf{p}_d(t)$. To this end, we first convert the system in Eq. (5.1) into a linear one using the feedback law

$$\mathbf{u} = \bar{\mathbf{M}}\,\mathbf{v} + \bar{\mathbf{C}}\,\dot{\mathbf{p}}, \tag{5.2}$$

where $\mathbf{v} \in \mathbb{R}^3$ is a new control input. Note that if we substitute Eq. (5.2) into (5.1) we obtain

$$\bar{\mathbf{M}}\,\ddot{\mathbf{p}} = \bar{\mathbf{M}}\,\mathbf{v}$$

and since $\bar{\mathbf{M}}$ is nonsingular this implies that

$$\ddot{\mathbf{p}} = \mathbf{v}, \tag{5.3}$$

so the system now exhibits double-integrator dynamics.

The system in Eq. (5.3) is easy to stabilize along $\mathbf{p}_d(t)$. Since its three scalar equations are uncoupled, it suffices to design a controller for each scalar subsystem independently. To do so, let us write the i-th component of Eq. (5.3) as

$$\ddot{p} = v \tag{5.4}$$

and let $p_d(t)$ be the desired trajectory for p. The first-order form of Eq. (5.4) is

$$\underbrace{\begin{bmatrix} \dot{p} \\ \ddot{p} \end{bmatrix}}_{\dot{\mathbf{z}}} = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} p \\ \dot{p} \end{bmatrix}}_{\mathbf{z}} + \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{\mathbf{B}} v,$$

which we compactly write as

$$\dot{\mathbf{z}} = \mathbf{A}\,\mathbf{z} + \mathbf{B}\,v \tag{5.5}$$

In $\mathbf{z}$-space, the trajectory to be followed, and its open-loop controls, are

$$\mathbf{z}_d(t) = \left(p_d(t), \dot{p}_d(t)\right),$$

$$v_d(t) = \ddot{p}_d(t),$$

and note that they satisfy

$$\dot{\mathbf{z}}_d = \mathbf{A}\,\mathbf{z}_d + \mathbf{B}\,v_d. \tag{5.6}$$

Then, the trajectory error is given by

$$\mathbf{e}(t) = \mathbf{z}(t) - \mathbf{z}_d(t) = \begin{bmatrix} p(t) - p_d(t) \\ \dot{p}(t) - \dot{p}_d(t) \end{bmatrix},$$

so its time derivative is

$$\dot{\mathbf{e}}(t) = \dot{\mathbf{z}}(t) - \dot{\mathbf{z}}_d(t). \tag{5.7}$$

By substituting Eqs. (5.5) and (5.6) into Eq. (5.7) we get

$$\dot{\mathbf{e}} = \mathbf{A}\,\underbrace{(\mathbf{z} - \mathbf{z}_d)}_{\mathbf{e}} + \mathbf{B}\,\underbrace{(v - v_d)}_{\bar{v}},$$

which shows that the error dynamics is given by

$$\dot{\mathbf{e}} = \mathbf{A}\,\mathbf{e} + \mathbf{B}\,\bar{v}. \tag{5.8}$$

Thus, the stabilization of the system in Eq. (5.5) along $\mathbf{z}_d(t)$ reduces to stabilizing $\mathbf{e}(t)$ to the origin of $\mathbb{R}^2$. This can be achieved by using the feedback law

$$\bar{v} = -\mathbf{K}\,\mathbf{e}, \tag{5.9}$$

where

$$\mathbf{K} = [k_p, k_v]$$

In this law, k_p and k_v are called the position and velocity gains, and it is well-known that, if they are positive,

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

irrespective of the initial condition $\mathbf{e}(0)$. Therefore, this controller ensures a globally-stable tracking of $p_d(t)$, which is quite a desirable property.

Observe that since $\bar{v} = v - v_d = v - \ddot{p}_d$, Eq. (5.9) can be expressed as

$$v = \ddot{p}_d - k_p\,(p - p_d) - k_v\,(\dot{p} - \dot{p}_d)\,, \tag{5.10}$$

which is the usual form assumed for a PD controller in trajectory tracking. By particularizing Eq. (5.10) for each one of the three coordinates in $\mathbf{p} = (x, y, \alpha)$, we see that the control law needed to stabilize the system in Eq. (5.3) along $\mathbf{p}_d(t)$ is

$$\mathbf{v} = \underbrace{\begin{bmatrix} \ddot{x}_d \\ \ddot{y}_d \\ \ddot{\alpha}_d \end{bmatrix}}_{\ddot{\mathbf{p}}_d} - \underbrace{\begin{bmatrix} k_{p,x} & 0 & 0 \\ 0 & k_{p,y} & 0 \\ 0 & 0 & k_{p,\alpha} \end{bmatrix}}_{\mathbf{K}_p} \underbrace{\begin{bmatrix} x - x_d \\ y - y_d \\ \alpha - \alpha_d \end{bmatrix}}_{\mathbf{p}-\mathbf{p}_d} - \underbrace{\begin{bmatrix} k_{v,x} & 0 & 0 \\ 0 & k_{v,y} & 0 \\ 0 & 0 & k_{v,\alpha} \end{bmatrix}}_{\mathbf{K}_v} \underbrace{\begin{bmatrix} \dot{x} - \dot{x}_d \\ \dot{y} - \dot{y}_d \\ \dot{\alpha} - \dot{\alpha}_d \end{bmatrix}}_{\dot{\mathbf{p}}-\dot{\mathbf{p}}_d} \tag{5.11}$$

where $k_{p,x}$, $k_{p,y}$, $k_{p,\alpha}$ and $k_{v,x}$, $k_{v,y}$, $k_{v,\alpha}$ are the position and velocity gains for x, y and α.

If we write Eq. (5.11) as

$$\mathbf{v} = \ddot{\mathbf{p}}_d - \mathbf{K}_p\,(\mathbf{p} - \mathbf{p}_d) - \mathbf{K}_v\,(\dot{\mathbf{p}} - \dot{\mathbf{p}}_d)$$

and substitute it in Eq. (5.2), we finally obtain

$$\mathbf{u} = \bar{\mathbf{M}}\,[\ddot{\mathbf{p}}_d - \mathbf{K}_p\,(\mathbf{p} - \mathbf{p}_d) - \mathbf{K}_v\,(\dot{\mathbf{p}} - \dot{\mathbf{p}}_d)] + \bar{\mathbf{C}}\,\dot{\mathbf{p}} \tag{5.12}$$

which is the desired computed-torque law for the Otbot. This law can also be written as

$$\mathbf{u} = \underbrace{\bar{\mathbf{M}}\,\ddot{\mathbf{p}}_d + \bar{\mathbf{C}}\,\dot{\mathbf{p}}}_{\mathbf{u}_{traj}} + \underbrace{\bar{\mathbf{M}}\left[-\mathbf{K}_p\left(\mathbf{p}-\mathbf{p}_d\right) - \mathbf{K}_v\left(\dot{\mathbf{p}}-\dot{\mathbf{p}}_d\right)\right]}_{\mathbf{u}_{corr}}, \tag{5.13}$$

where $\mathbf{u}_{traj}$ is the nominal torque needed to follow $\mathbf{p}_d(t)$ when we are on track along this trajectory, and $\mathbf{u}_{corr}(t)$ is the correction torque used to compensate position and velocity errors.

Note that to control the robot using this law we need feedback of the $\mathbf{p}$ variables x, y, and α, as well as the angle φ_p of the pivot joint. The $\mathbf{p}$ variables are needed to compute the position and velocity errors, and the pivot joint angle is used together with α to evaluate $\bar{\mathbf{M}}$ and $\bar{\mathbf{C}}$. Such a feedback can be obtained from the onboard IMU and the angular encoder of the pivot joint.

5.2 Tuning of the control law

Once the controller has been defined, we must set appropiate values for k_p and k_v in each instance of Eq. (5.10). To this end we substitute Eq. (5.9) into Eq. (5.8) to get

$$\dot{\mathbf{e}} = \underbrace{\left(\mathbf{A}-\mathbf{B}\mathbf{K}\right)}_{\mathbf{C}}\mathbf{e},$$

which is the autonomous ODE describing the error of the closed-loop system for the p variable in consideration. From the theory of linear systems we know that the solution of this ODE is

$$\mathbf{e}(t) = \underbrace{e^{s_1 t}c_1\mathbf{v}_1}_{\mathbf{e}_1(t)} + \underbrace{e^{s_2 t}c_2\mathbf{v}_2}_{\mathbf{e}_2(t)}, \tag{5.14}$$

where s_1 and s_2 are the eigenvalues of $\mathbf{C} = \mathbf{A} - \mathbf{B}\mathbf{K}$, $\mathbf{v}_1$ and $\mathbf{v}_2$ are their associate eigenvectors, and c_1 and c_2 are the constants that satisfy

$$\begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \mathbf{e}(0).$$

To tune the controller, we will select values for s_1 and s_2 that ensure an overdamped exponential decay of $\mathbf{e}(t)$, and then we will find the gains k_p and k_v that correspond to such values. These gains can be written as a function of s_1 and s_2 by noting that s_1 and s_2 are the roots of

$$\det\left(\mathbf{C} - \lambda\mathbf{I}\right) = \lambda^2 + k_v\lambda + k_p, \tag{5.15}$$

or equivalently of

$$(\lambda - s_1)(\lambda - s_2) = \lambda^2 - \lambda(s_1 + s_2) + s_1\, s_2, \tag{5.16}$$

so identifying the right-hand sides of (5.15) and (5.16) we obtain

$$\begin{aligned} k_p &= s_1\, s_2, \\ k_v &= -(s_1 + s_2). \end{aligned}$$

To force an overdamped decay of $\mathbf{e}(t)$ towards zero, s_1 and s_2 must be negative real numbers. At this point it is customary to choose

$$s_1 = -\frac{4}{T_{stab}},$$

where T_{stab} is the time we allow for $\mathbf{e}(t)$ to be smaller than $2\%\ \mathbf{e}(0)$, and

$$s_2 = 10\ s_1,$$

so in Eq. (5.14) $\mathbf{e}_2(t)$ decays much more rapidly than $\mathbf{e}_1(t)$. In doing so we have

$$\mathbf{e}(t) = e^{-\frac{4}{T_{stab}}t} c_1\mathbf{v}_1 + e^{-\frac{40}{T_{stab}}t} c_2\mathbf{v}_2,$$

so for $t = T_{stab}$ we obtain

$$\mathbf{e}(T_{stab}) = \underbrace{e^{-4}}_{0{,}018} c_1\mathbf{v}_1 + \underbrace{e^{-40}}_{\simeq 0} c_2\mathbf{v}_2,$$

which ensures that $\mathbf{e}(T_{stab}) \leqslant 2\%\ \mathbf{e}(0)$ as desired. Note that $\mathbf{e}(t)$ contains both the position and velocity errors, so both errors will almost disappear after T_{stab} seconds.

In the Otbot we have set $T_{stab} = 3\text{s}$ for all variables $p \in \{x, y, \alpha\}$, which yields

$$\begin{aligned} s_1 &= -\frac{4}{3} = -1.333, \\ s_2 &= 10\ s_1 = -13.333, \end{aligned}$$

and,

$$\begin{aligned} k_p &= s_1\, s_2 = 17.778, \\ k_v &= -(s_1 + s_2) = 14.667. \end{aligned}$$

Therefore, the gains we have used in Eq. (5.11) are

$$\mathbf{K}_p = \begin{bmatrix} 17.778 & 0 & 0 \\ 0 & 17.778 & 0 \\ 0 & 0 & 17.778 \end{bmatrix}, \quad \mathbf{K}_v = \begin{bmatrix} 14.667 & 0 & 0 \\ 0 & 14.667 & 0 \\ 0 & 0 & 14.667 \end{bmatrix}.$$

Note however that it is possible to choose different stabilitzation times T_{stab} for each variable if desired, which would yield different gains along the diagonals of $\mathbf{K}_p$ and $\mathbf{K}_v$.

5.3 Checking the feasibility of the control torques

Once the gains $\mathbf{K}_p$ and $\mathbf{K}_v$ have been decided, it is important to check that the control inputs given by Eq. (5.13),

$$\mathbf{u} = \underbrace{\bar{\mathbf{M}}\,\ddot{\mathbf{p}}_d + \bar{\mathbf{C}}\,\dot{\mathbf{p}}}_{\mathbf{u}_{traj}} + \underbrace{\bar{\mathbf{M}}\,[-\mathbf{K}_p\,(\mathbf{p}-\mathbf{p}_d) - \mathbf{K}_v\,(\dot{\mathbf{p}}-\dot{\mathbf{p}}_d)]}_{\mathbf{u}_{corr}},$$

stay within the allowable torque limits of the motors along $\mathbf{p}_d(t)$, otherwise the global stability properties of the controller may be compromised.

The exact value of $\mathbf{u}_{traj}(t)$ and $\mathbf{u}_{corr}(t)$ cannot be known a priori, as they depend on unmodelled perturbations that occur during the actual operation of the robot. However, we may find reasonable bounds for them if we know the intervals of variation for $\mathbf{e}_p = \mathbf{p} - \mathbf{p}_d$ and $\mathbf{e}_v = \dot{\mathbf{p}} - \dot{\mathbf{p}}_d$. These bounds can be computed as follows.

We first simulate the desired trajectory $\mathbf{p}_d(t)$ in open loop, using $\mathbf{u} = \mathbf{u}_{traj}(t)$ as the control input. This will provide the robot state $\mathbf{x}(t) = (\mathbf{q}(t), \dot{\mathbf{q}}(t))$ for each time t along $\mathbf{p}_d(t)$, so $\dot{\mathbf{p}}(t)$, $\bar{\mathbf{M}}(t)$ and $\bar{\mathbf{C}}(t)$ will also be known. The real trajectory will be slightly different, but we can assume that the actual values $\dot{\mathbf{p}}(t)$, $\bar{\mathbf{M}}(t)$ and $\bar{\mathbf{C}}(t)$ stay close to those on $\mathbf{p}_d(t)$.

For each time t we then evaluate the range of possible values taken by $\mathbf{u}_{corr}(t)$ and $\mathbf{u}_{traj}(t)$ using interval arithmetics [14]. Since $\bar{\mathbf{M}}$, $\mathbf{K}_p$ and $\mathbf{K}_v$ are all known at time t, the expression of $\mathbf{u}_{corr}(t)$ is linear in $\mathbf{e}_p$ and $\mathbf{e}_v$, and it is easy to obtain an interval bound for $\mathbf{u}_{corr}(t)$ using sums of intervals

$$[x_{min}, x_{max}] + [y_{min}, y_{max}] = [x_{min} + y_{min}, x_{max} + y_{max}],$$

and multiplications of an interval by a constant

$$k \cdot [x_{min}, x_{max}] = \begin{cases} [k \cdot x_{max}, k \cdot x_{min}] & for\ k < 0 \\ [k \cdot x_{min}, k \cdot x_{max}] & for\ k \geq 0 \end{cases}.$$

Regarding $\mathbf{u}_{traj}(t)$, since $\mathbf{e}_v$ takes values within a prescribed interval and $\dot{\mathbf{p}}_d(t)$ is known, the interval of variation of $\dot{\mathbf{p}}(t)$ will also be known, which allows the interval evaluation of $\bar{\mathbf{C}}\ \dot{\mathbf{p}}$ using the previous operations. The resulting intervals must be shifted using the scalar quantities of $\bar{\mathbf{M}}(t)\ \ddot{\mathbf{p}}_d(t)$ to obtain $\mathbf{u}_{traj}(t)$.

In the end, we only have to add the intervals obtained for $\mathbf{u}_{traj}(t)$ and $\mathbf{u}_{corr}(t)$ for each time t, and check whether the result stays within the allowable torque limits of the robot.

5.4 Test cases

We next explain several simulations that we have done to verify the obtained control law. We start by doing some experiments in order to test the omnidirectionality of the robot (Section 5.4.1). Some experiments are then performed to verify that the controller has the ability to stabilize trajectories that are the result of a motion planner (Section 5.4.2). Finally, we verify the effectiveness of the control law by simulating the system under unmodelled force perturbations (Section 5.4.3). In all experiments we have used the same robot parameters assumed in Chapter 4. Viscous friction has been neglected, as it does not bring anything illustrative to our tests.

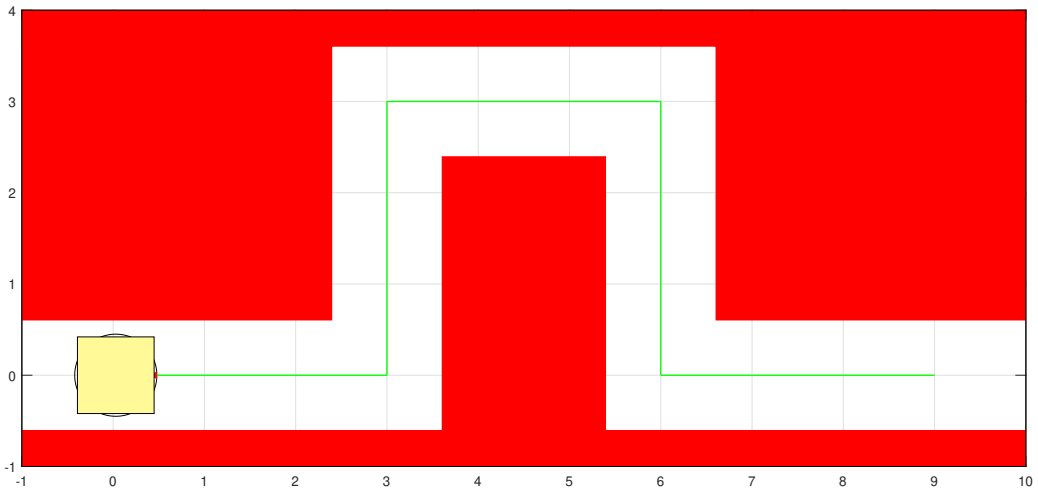


Figure 5.1: Narrow corridor experiment. The vehicle has to transport the yellow load at a constant orientation with a constant speed of $0.6\mathrm{m/s}$ along the green path. See https://youtu.be/dhKDoDd0wZo for an animated version of the motion. The motion underlying the chassis can be appreciated in https://youtu.be/c62ks498Z2U.

5.4.1 Narrow corridor test

As we explained in Chapter 1, one of the main advantages of the Otbot is that its platform has omnidirectional mobility in the plane. This is a very desirable property that allows a simpler negotiation of obstacles and narrow corridors, or the correction of small errors without having to maneuver. To test our controller in one of these situations we have set the Otbot to transport a load along a narrow corridor with 90° turns while keeping $\alpha = 0^\circ$ (see Fig. 5.1 and its video animations). Notice that such a movement would be impossible for a vehicle with, e.g., Ackerman kinematics, or for a standard differential drive robot. Even if the load were allowed to rotate along the path, the amount of maneuvering would be considerable at the corners of the corridor.

In this test, the path to be followed consists of five rectilinear segments of $3\mathrm{m}$ in length, which have to be traversed at a constant velocity of $0.6\mathrm{m/s}$. Thus, each segment has to be covered in 5 seconds by the robot, which is assumed to be at rest at the beginning. The tracking errors obtained in this example are given in Figs. 5.2 and 5.3.

Notice that, because the robot is initially at rest, errors in both x and $\dot{x}$ appear initially. However, the robot is able to counteract them in $T_{stab} = 3\mathrm{s}$ as we assumed in Section 5.2. This point is easier to check in the error plot for $\dot{x}$ (Fig. 5.3, top).

Note also that errors in x, y, $\dot{x}$, and $\dot{y}$ arise at each 90° corner of the path because of the sudden change in direction of the reference velocity (x_d, y_d). Again, the robot takes about $3\mathrm{s}$ to eliminate this error.

From the plots we see that the error in the platform angle α is negligible, so the platform stays with $\alpha = 0^\circ$ during the motion. This is in agreement with the fact that the robot departs from $\alpha = 0^\circ$ for $t = 0\mathrm{s}$, and the chassis cannot transmit any torque to the platform because viscous friction is assumed to be negligible.

In the last 5 seconds of the trajectory the reference signal for $x(t)$ and $y(t)$ is kept at

$$x_d(t) = 9\mathrm{m}$$
$$y_d(t) = 0\mathrm{m}$$

which explains the positive error in x that we observe, and its progressive elimination. Thus, the robot slightly overshoots the goal position $(9, 0)\mathrm{m}$, and has to return backwards to attain it. The overshoot can be eliminated by simply designing a trajectory that smoothly achieves $(\dot{x}(t), \dot{y}(t)) = (0, 0)$ for $t = 25\mathrm{s}$.

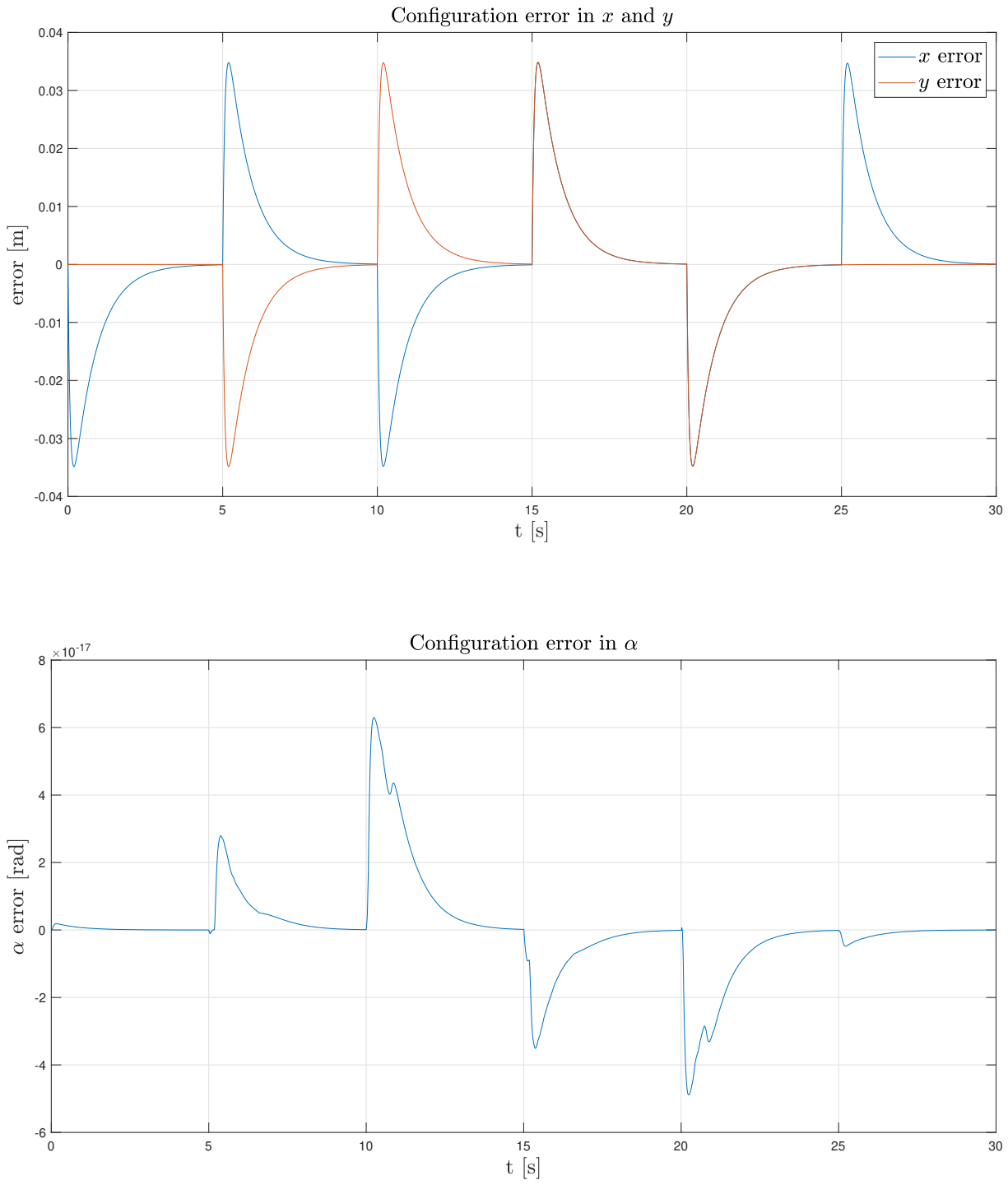


Figure 5.2: Configuration errors in x, y and α in the narrow corridor example.

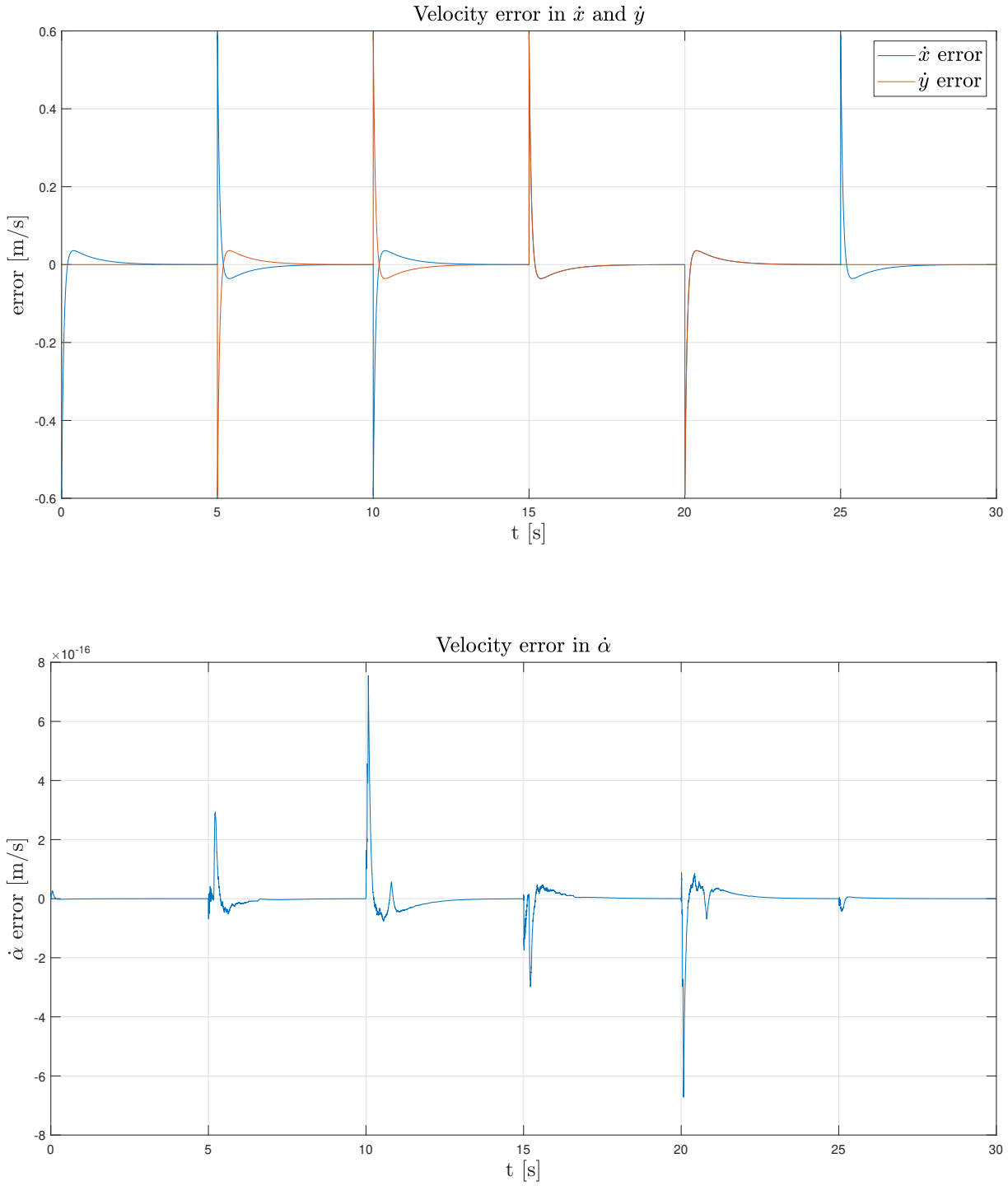


Figure 5.3: Velocity errors in $\dot{x}$, $\dot{y}$ and $\dot{\alpha}$ in the narrow corridor example.

5.4.2 Tracking the trajectory of a motion planner

In this section we show another practical example of the usage of the controller. It is well known that one way to design robot trajectories is by means of a motion planner. Such a planner allows us to obtain trajectories that satisfy multiple constraints on the states and actions of the robot, like collision-avoidance, or velocity or torque-limit constraints. In this particular example we wish to stabilize a trajectory that was obtained with the motion planner in [15], which generates the trajectory by discretizing $x(t)$ and $u(t)$, and solving a constrained optimization problem. Because of the discretization, the planner returns a sequence of actions $\mathbf{u}_0, \ldots, \mathbf{u}_N$, and states $\mathbf{x}_0, \ldots, \mathbf{x}_N$ that are not exactly compatible. That is, if we use $\mathbf{u}_0, \ldots, \mathbf{u}_N$ to perform open-loop control, the robot will end up deviating from the desired trajectory, possibly colliding with some obstacle. Instead, if we use the controller to directly track the states $\mathbf{x}_0, \ldots, \mathbf{x}_N$ in closed loop, we will see that the robot is able to accurately follow the desired trajectory.

The planned trajectory we wish to stabilize is shown in green in Fig. 5.4 top (units in m). Here, the robot is set to traverse a narrow corridor that is crowded with circular obstacles. The figure also shows as a dashed line the trajectory that results from controlling the robot in open loop using $\mathbf{u}_0, \ldots, \mathbf{u}_N$. From the associate video we note that the robot deviates from the green trajectory and collides with two obstacles. The rest of Fig. 5.4 provides the position and velocity errors obtained during the motion, which are significant, and even cumulative in x and y. In contrast, Fig. 5.5 shows how, when tracking $\mathbf{x}_0, \ldots, \mathbf{x}_N$ in closed loop, the robot can accurately follow the desired trajectory. Position and velocity errors are smaller than $1\mathrm{mm}$ and $1\mathrm{cm/s}$ most of the time, while those in α and $\dot{\alpha}$ are below $0.001\mathrm{rad}$ and $0.005\mathrm{rad/s}$ in general. The errors we obtain are not zero exactly because the reference trajectories for $\dot{x}$, $\dot{y}$, and $\dot{\alpha}$ are obtained by computing finite differences on the discrete trajectories for x, y, and α. Thus, the position and velocity reference signals have small mutual inconsistencies.

5.4.3 Disturbance rejection

We next want to test the robustness of the control law to external force perturbations not included in the model. To do so, we will simulate the effect of applying a small external force $\mathbf{f}_p = (f_{p,x}, f_{p,y})$ on the pivot joint P of the platform, at various time instants during the simulation horizon.

To simulate the effect of these forces, the generalized force corresponding to $\mathbf{f}_p$ must be added to the right-hand side of Eq. (3.1) This force, which we denote by $\mathbf{Q}_p$, is obtained analogously to how the generalized force of actuation was obtained (Section 3.1.3). Clearly,

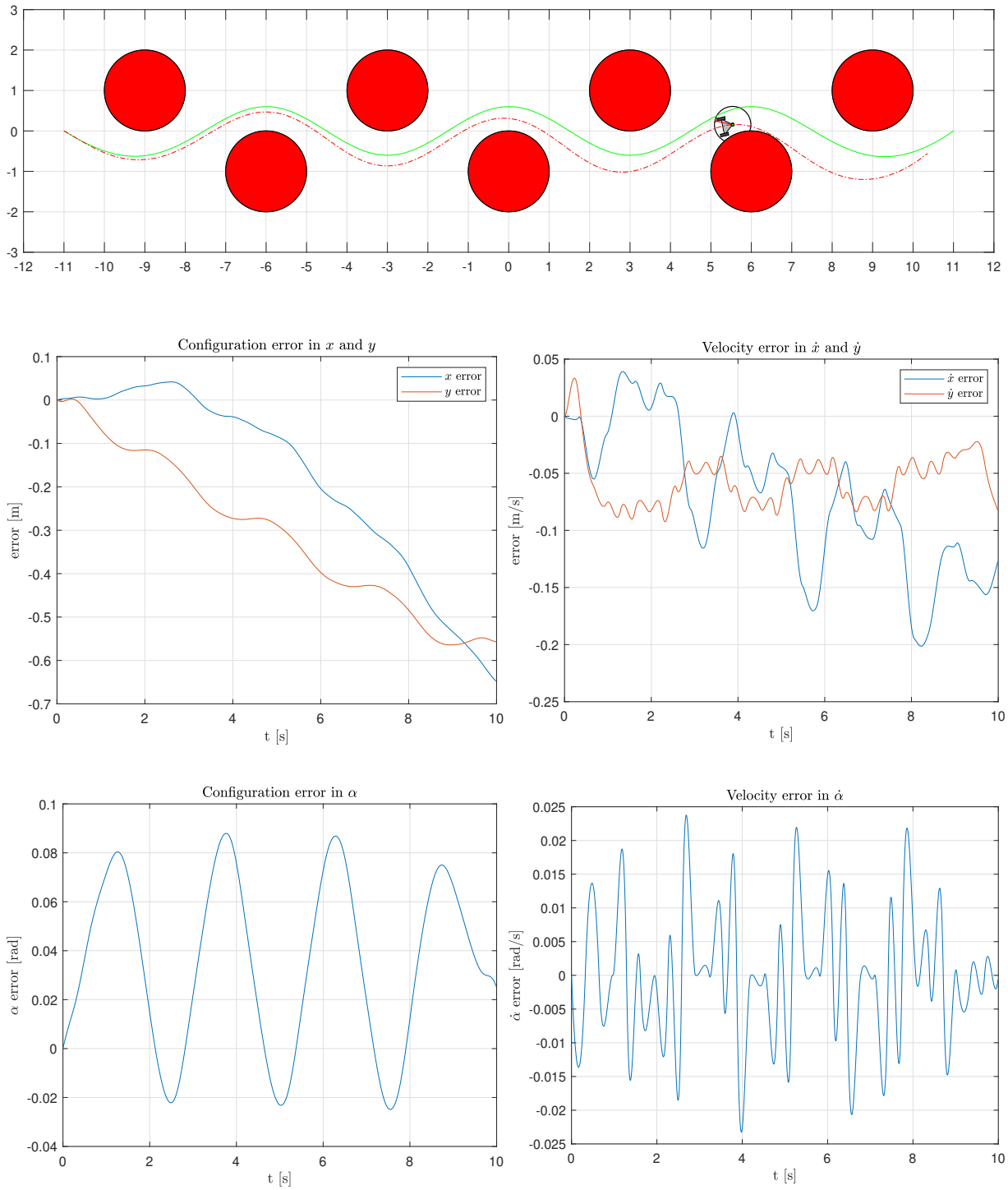


Figure 5.4: Top: Simulation of the crowded corridor experiment in open-loop. The vehicle has to pass through the corridor without colliding with the circular obstacles (the axes' units are in m). The desired and obtained trajectories are shown in green and red respectively. Bottom: The error plots of the motion. See https://youtu.be/J3dnIHSabnA for an animated version of the experiment.

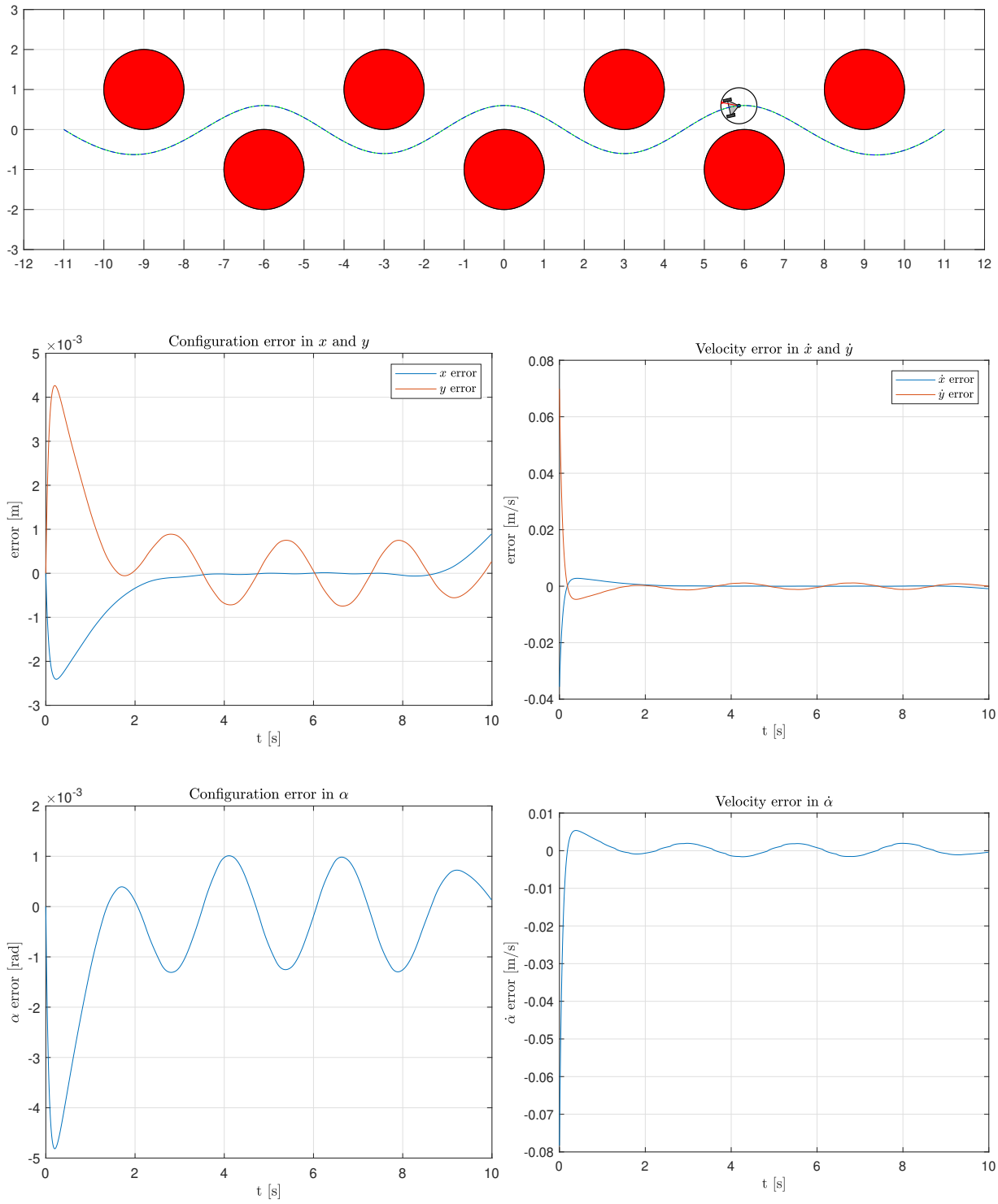


Figure 5.5: Top: Simulation of the crowded corridor experiment in closed-loop. The desired and obtained trajectories now coincide, and the robot is able to avoid the obstacles. Bottom: The error plots of the motion. See https://youtu.be/x5mmnUQ9x50 for an animated version of the experiment.

the virtual power generated by $\mathbf{f}_p$ is

$$P_p = \begin{bmatrix} f_{p,x} & f_{p,y} \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{5.17}$$

which, to identify $\mathbf{Q}_p$, must be written in the form

$$P_p = \mathbf{Q}_p^{\mathsf{T}}\, \dot{\mathbf{q}}. \tag{5.18}$$

However, since

$$\dot{\mathbf{q}} = (\dot{x}, \dot{y}, \dot{\alpha}, \dot{\varphi}_r, \dot{\varphi}_l, \dot{\varphi}_p)\,,$$

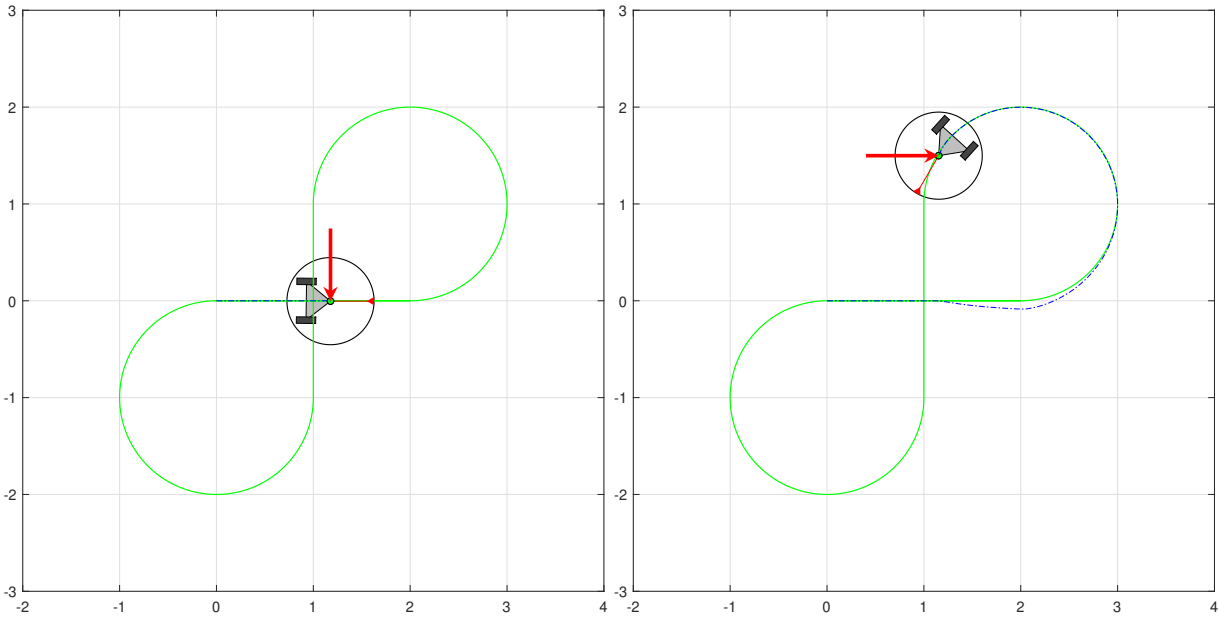
we clearly see that

$$\mathbf{Q}_p = (f_{p,x}, f_{p,y}, 0, 0, 0, 0)\,.$$

To illustrate the effect of these forces, the results of their application during a simulation of 18s will be shown below (Fig. 5.6). During this simulation, we will see the Otbot following a "figure 8" trajectory composed of straight and circular sections. During this tracking three perturbation forces are applied on F:

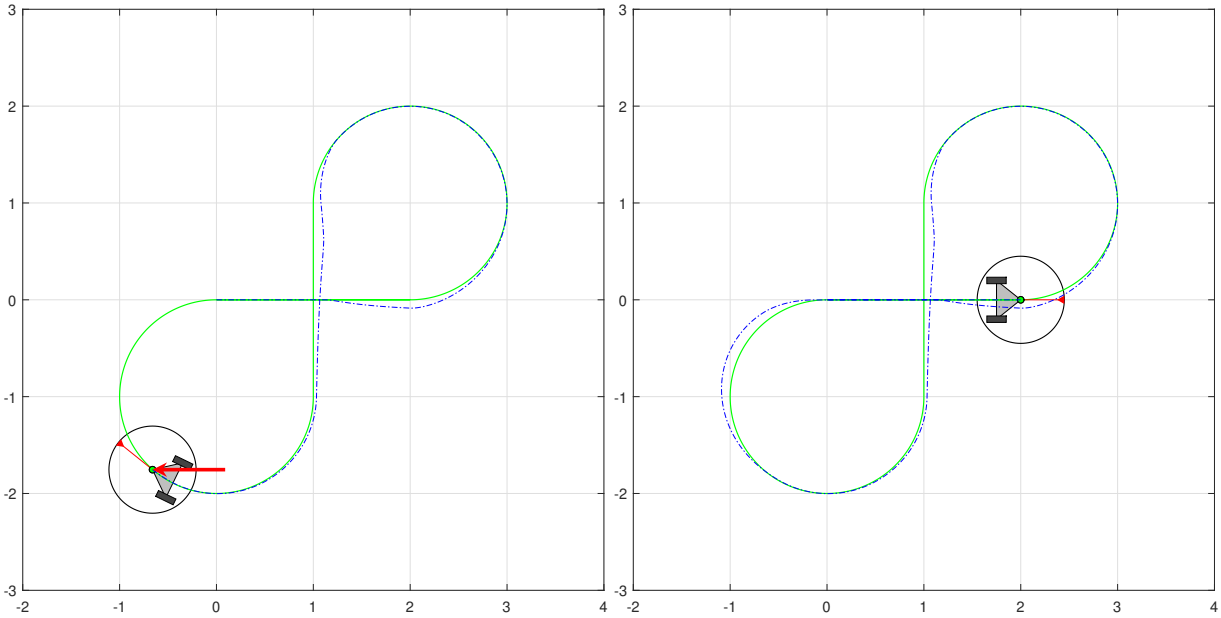
- A force of 150N in the negative y direction at $t = 4\text{s}$.
- A force of 200N in the positive x direction at $t = 8\text{s}$.
- A force of 350N in the negative x direction at $t = 11\text{s}$.

The three forces are applied individually during 1s from the times indicated. From the plots in Figs. 5.7 and 5.8 and the video in https://youtu.be/6bkYEmZqX_o we can see how the Otbot corrects all the deviations caused by the three perturbation forces in about 3s as expected.



(a) First disturbance applied at 3s.

(b) Second disturbance applied at 8s.



(c) Third disturbance applied at 11s.

(d) Otbot at the goal position.

Figure 5.6: Disturbance rejection test. The vehicle has to follow the green path, but at certain time intervals we apply force disturbances which deviate the vehicle from the desired trajectory. The actual trajectory path is marked in blue. See https://youtu.be/6bkYEmZqX_o for an animated version of the motion.

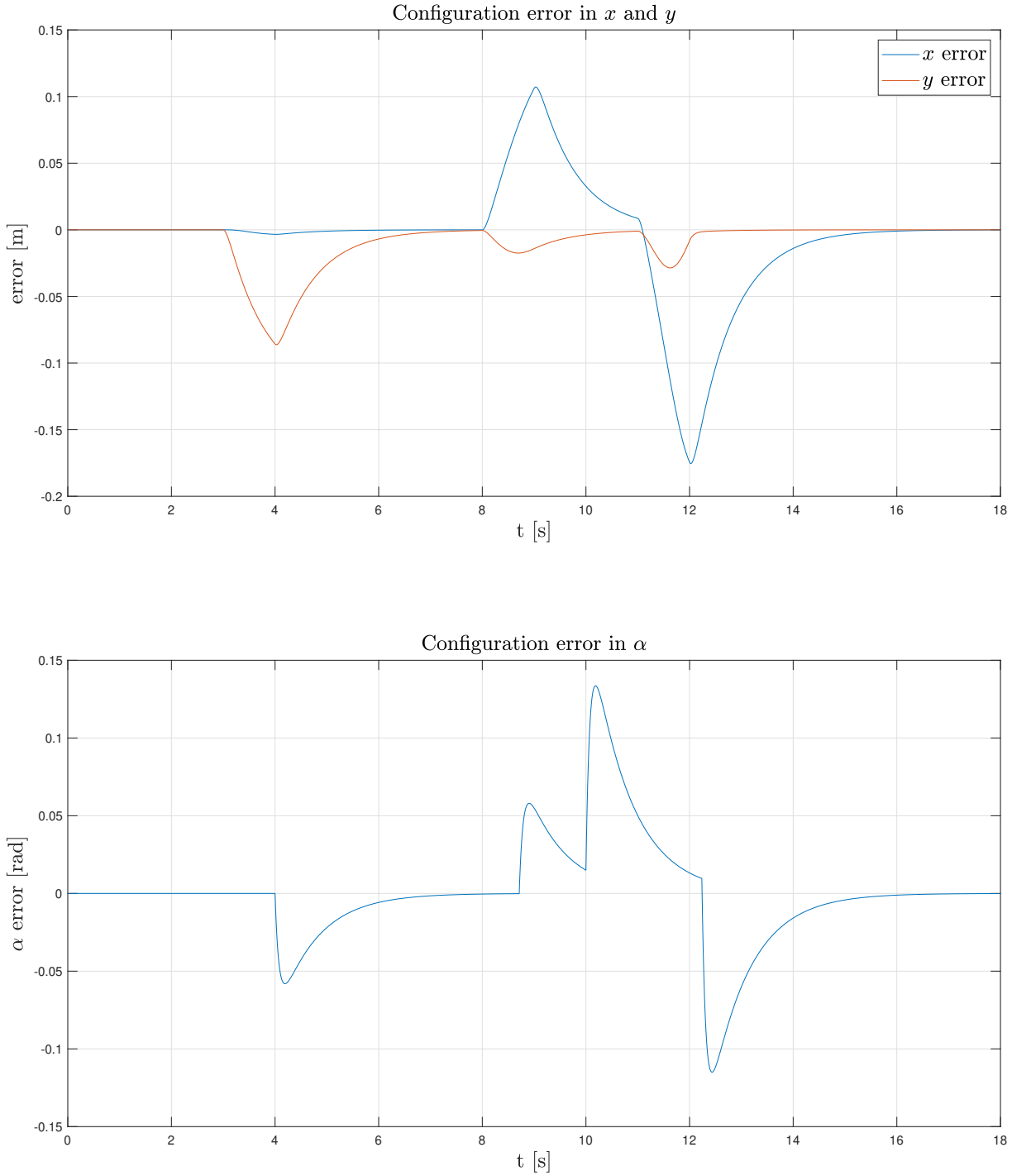


Figure 5.7: Configuration errors in x, y, α in the disturbance rejection test.

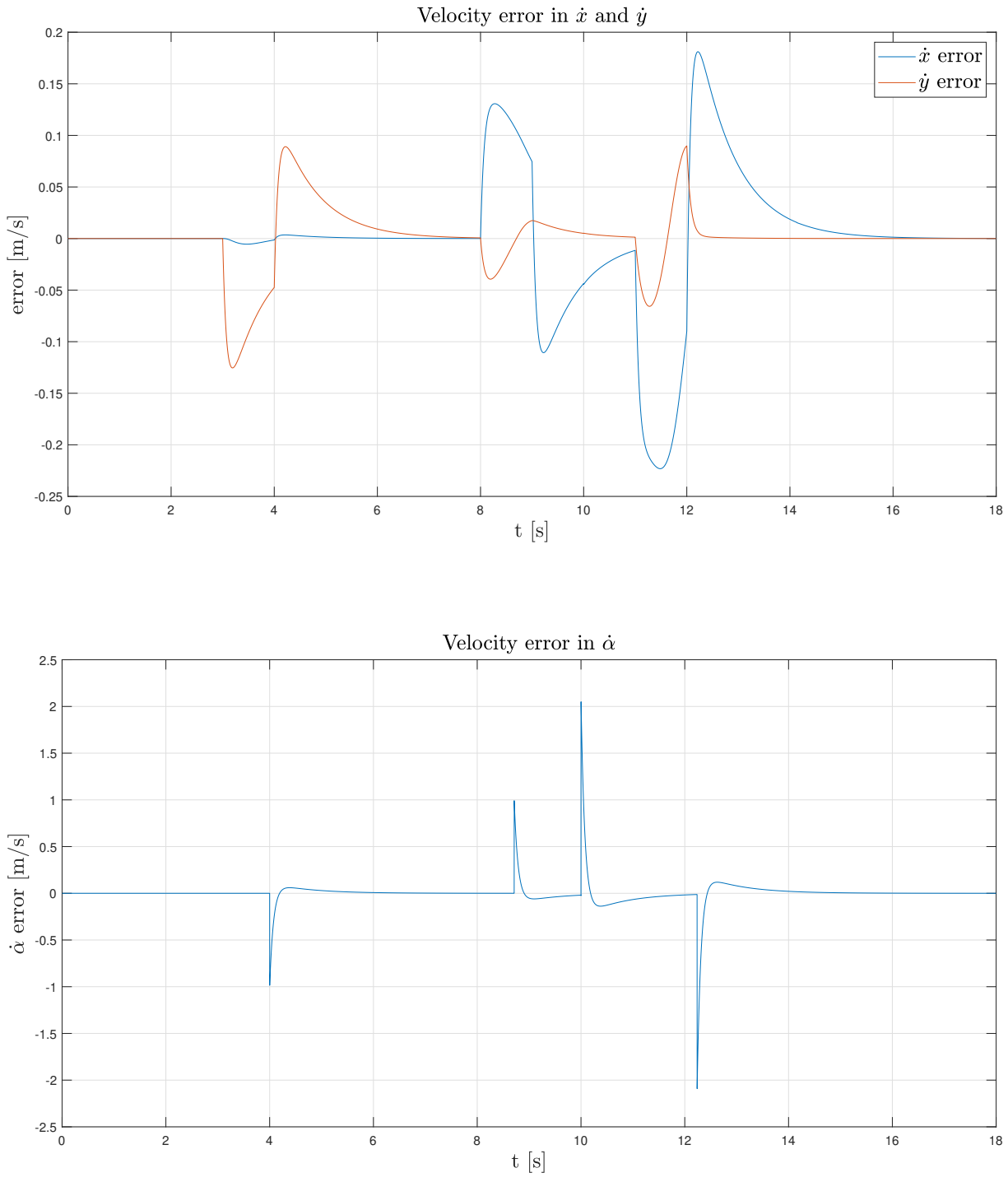


Figure 5.8: Velocity errors in $\dot{x}$, $\dot{y}$, $\dot{\alpha}$ in the disturbance rejection test.

6
Conclusions

The following is a summary of the main contributions of this work, and some points that could be addressed in future extensions.

6.1 Contributions

In this project we have obtained a dynamic model for the Otbot, we have proposed a method to identify its parameters, and we have defined a control law that allows the tracking of pre-established trajectories in a robust and accurate way.

The dynamic model is very generic. It takes into account the mass and moment of inertia of all robot bodies, the viscous friction on the axes, and assumes arbitrary positions for the centers of mass of the chassis and the platform. The model has also been manipulated to obtain compact solutions of the direct and inverse dynamic problems, which are necessary to simulate the robot motions or to obtain the torques that generate a certain trajectory. We have also obtained the dynamic model reduced to the task space, which is key to design the control law we propose.

The identification method is based on exciting the robot with constant torques for a few seconds, recording the signals of certain sensors of the robot, and determining the values of the parameters that minimize the difference between the predicted signals and the recorded ones. This minimization must be done on the basis of a fairly good estimate of the actual parameters, but the process has fairly large basins of attraction around them. The identification experiment, moreover, uses robot sensors exclusively, and the learning motion can be performed in very small spaces of, only about $4\mathrm{m}^2$. Therefore, there is no need to resort to special infrastructure external to the robot to identify its model.

The controller obtained is based on the computed torque technique, thus enjoying global stability. This means that the robot will converge to the desired trajectory regardless of its initial conditions, as long as the applied torques do not exceed the limits of the motors themselves. The

controller's gains can also be easily adjusted to ensure that tracking errors are eliminated within a pre-set time. Finally, the controller has been validated in simulation by applying it to track (1) a narrow corridor where the robot's omnidirectional capability must be used to make sudden changes of direction, (2) a motion with small inconsistencies between position and velocity references, and (3) a trajectory during which unmodelled force disturbances appear. In all cases the controller eliminates the errors that arise in the desired stabilization time.

6.2 Future work

The Otbot is a mobile robotic platform that, despite being little known, has interesting properties. The fact that it achieves omnidirectionality with conventional wheels makes it advantageous compared to vehicles using Mecanum or omni wheels, which are more complex to manufacture and maintain, and support lower loads in general. The Otbot also does not exhibit actuation singularities at any point in its state space, which makes it preferrable to other omnidirectional platforms with conventional wheels that present such critical configurations [16]. For all these reasons, we believe that it would be good to continue studying this robot in the future, to address the following aspects among others:

- A detailed study should be made of the advantages that the Otbot may have compared to other non-omnidirectional platforms that are widely used in the industry, such as Amazon's Kiva robot [17]. These robots resemble the Otbot, but the pivot joint is on the wheels' axes, so they do not have omnidirectional capability. While the Otbot can move continuously through narrow corridors with sudden changes of direction, the Kiva robots may need to stop, reorient, and continue in order to make progress. It would be good to compare the minimum-time trajectories of the two robots and determine how faster is Otbot in comparison.
- An experimental prototype of the Otbot would have to be built, equipped with the necessary sensors in order to experimentally validate the parameter identification and control methods we propose.
- To validate the identification process one should identify the parameters using the learning trajectory we propose, and then see how well these parameters predict other movements of the robot. This check could be done by recording new sensory signals with movements generated by known control actions, and seeing whether the identified model anticipates these movements from the mentioned actions.
- To validate the control law, we should see whether the robot is able to follow the desired trajectory despite small disturbances or wheel slippages that could affect it. This means

that good measurements of the position and orientation of the platform would have to be available, which could be achieved by means global positioning systems, or inertial navigation systems complemented by positioning based on calibrated landmarks.

- The robot could be equipped with a path planner to provide it with autonomy to generate navigation plans for obstacle environments. The work in [15] is an important step forward in this direction.
- Finally, the proposed controller could be made adaptive, so the system itself could adjust the dynamic parameters if they changed on the fly, such as when depositing a load of unknown mass and moment of inertia on an arbitrary point on the platform.

Bibliography

[1] J. Agulló, S. Cardona, and J. Vivancos, "Kinematics of vehicles with directional sliding wheels," *Mechanism and Machine Theory*, vol. 22, pp. 295–301, jan 1987.

[2] J. Agulló, S. Cardona, and J. Vivancos, "Dynamics of vehicles with directionally sliding wheels," *Mechanism and Machine Theory*, vol. 24, pp. 53–60, jan 1989.

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

[4] M.-J. Jung, H.-S. Shim, H.-S. Kim, and J.-H. Kim, "The miniature omnidirectional mobile robot OmniKity-I (OK-I)," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, IEEE, 1999.

[5] M.-J. Jung, H.-S. Kim, S. Kim, and J.-H. Kim, "Omnidirectional mobile base OK-II," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, IEEE, 2000.

[6] M.-J. Jung and J.-H. Kim, "Mobility augmentation of conventional wheeled bases for omnidirectional motion," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 1, pp. 81–87, 2002.

[7] R. M. Murray, Z. Li, and S. Sastry., *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994.

[8] L. Ljung., *System identification: theory for the user*. Prentice-Hall 1999., 1999.

[9] J. J. Moré and D. Sorensen, "Computing a trust region step," *Journal on Scientific and Statistical Computing*, vol. 3, pp. 553–572, 1983.

[10] J. Nocedal and S. Wright, *Numerical optimization*. Springer, 2006.

[11] MATLAB Manual, "Trust region reflective least squares." https://bit.ly/2OxzHVa.

[12] MATLAB Manual, "Estimate nonlinear grey-box model parameters." https://bit.ly/2PFTsut.

[13] J. Dormand and P. Prince, "A family of embedded Runge-Kutta formulae," *Appl. Math.*, vol. 6, pp. 19–26, 1980.

[14] L. Jaulin, M. Kieffer, O. Didrit, and É. Walter, *Applied Interval Analysis*. Springer London, 2001.

[15] M. Gautier, “Planning and Control of Optimal Trajectories for an Omnidirectional Tire-wheeled Robot,” Master’s thesis, École Polytechnique Fédérale de Lausanne - EPFL, 2021.

[16] J. A. Batlle and A. Barjau, “Holonomy in mobile robots,” *Robotics and Autonomous Systems*, vol. 57, no. 4, pp. 433–440, 2009.

[17] R. D’Andrea and P. Wurman, “Future challenges of coordinating hundreds of autonomous vehicles in distribution facilities,” in *2008 IEEE International Conference on Technologies for Practical Robot Applications*, pp. 80–83, IEEE, 2008.

A
Source code

In order to be able to estimate the parameters, design the control system and make the corresponding simulations, a number of functions and *scripts* have been created in MATLAB. As mentioned in previous chapters, we will use the symbolic toolbox of MATLAB to obtain the kinematic and dynamic models of the Otbot. To make the use of this toolbox more visual we have used *liveSripts*, which allow the inclusion of text and images interspersed with parts of the code. These files are in `.mlx` format and will also be shown in this appendix.

This appendix is divided into four sections. Section A.1 provides the necessary scripts to obtain the kinematic and dynamic symbolic models of the Otbot. Section A.2 details the code that uses these models to numerically simulate the behavior of the robot. Section A.3 provides the algorithms used to validate the parameter estimation process in its various steps. Section A.4 finally provides the code used to implement and test the computed-torque controller of the robot. While the first section is implemented using Matlab livescripts, the latter three sections are written in common Matlab language. Both Sections A.3 and A.4 use functions defined earlier in Section A.2.

In parallel to the main scripts and functions a group of common functions has been designed that can be called by any of the previous groups of functions or scripts. These functions are responsible for solving secondary aspects of the simulation. For example, it is necessary to draw the configuration of the robot from the vector of generalized coordinates, so we have a specific group of functions that is responsible to implement this task. This functions group is called multiple times from the main script and in this way it is possible to generate a video of the simulated movement. There are also parts of the code dedicated to generating and saving plots of results, and other parts responsible for drawing the trail of the robot's trajectory throughout the animations.

A.1 Kinematic and dynamic modelling programs

Here we display the basic scripts that sustain all the work done in this project. There are a total of three. The first one named `kinematic_model.mlx`, is in charge of building all the matrices and vectors that take part in the Otbot's direct and inverse instantaneous kinematics. The second one, called `dynamic_model.mlx` and builds the main matrices and vector expressions that we will need for the dynamic model of the robot. Finally, the third script, called `dynamic_model_task_space.mlx`, analogous to the first two, is responsible for obtaining the elements that make up the dynamic model in task space, which is necessary to subsequently

simplify the model and develop the controller. All of them have been written using MATLAB's *liveScripts* as they allow to combine code with formal explanatory text and images. In order to get the simulations right in the first instance, these scripts need to be run to generate a series of `.mat` files to be used in the other parts of the code. Once these `.mat` files have been generated, there is no need to run the *liveScripts* again, as we can read the information directly from the files.

Kinematic model of Otbot

This live script develops the kinematic model of the Otbot.

Table of Contents

Initializations:

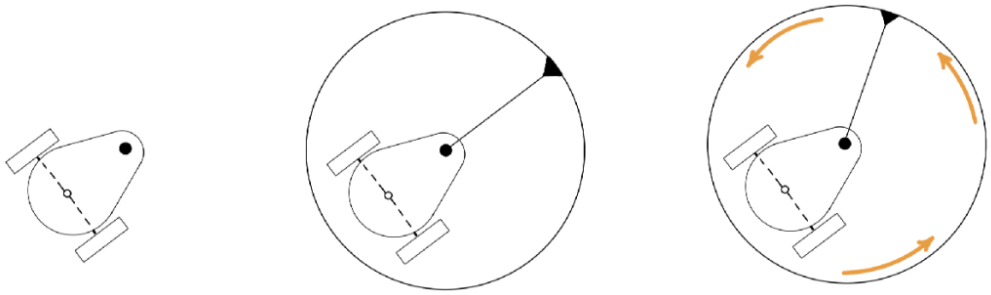
```
% Clear all variables, close all figures, and clear the command window
clearvars
close all
clc

% Start stopwatch
tic

% Display the matrices with rectangular brackets
sympref('MatrixWithSquareBrackets',true);

%  Avoid Matlab's own substitution of long expressions
sympref('AbbreviateOutput',false);

% Symbolic variables to be used (see the figures and explanations below)
syms l_1 l_2 r          % Pivot offset, semiaxis length, and wheel radius
syms theta              % Absolute angle of the chassis
syms theta_dot          % Absolute angular velocity of the chassis
syms v                  % Velocity of the chassis point M along 1'

syms varphi_dot_l       % Angular velocity of left wheel
syms varphi_dot_r       % Angular velocity of right wheel

syms a b                % Absolute coordinates of M
syms a_dot b_dot        % Absolute velocity components of M
```

```
syms x y                % Absolute coordinates of the pivot point P
syms x_dot y_dot        % Absolute velocity components of P
syms alpha              % Absolute angle of the platform
syms alpha_dot          % Absolute angular velocity of the platform

syms varphi_p           % Pivot angle
syms varphi_dot_p       % Angular velocity of the pivot joint
```

1. Introduction

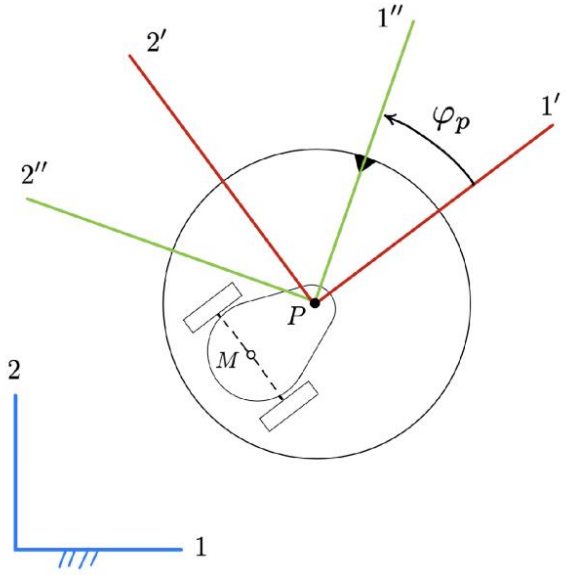
Reference frames and vector bases

The following pictures describe the kinematic structure of Otbot:



The robot chassis (left picture) is a classical differential drive robot whose two wheels are powered by DC motors. Passive caster wheels below the chassis (not drawn) impede the tumbling of the robot. A circular platform is mounted on top of the chassis (center picture) which can rotate relative to it (right picture). The rotation is performed by means of a pivot joint (black dot) that is actuated by another DC motor. Note that the pivot joint has an offset relative to the wheels' axis. This offset will allow the circular platform to behave like an omnidirectional vehicle in the plane.

To obtain the kinematic and dynamic models of the robot, we use the following reference frames:

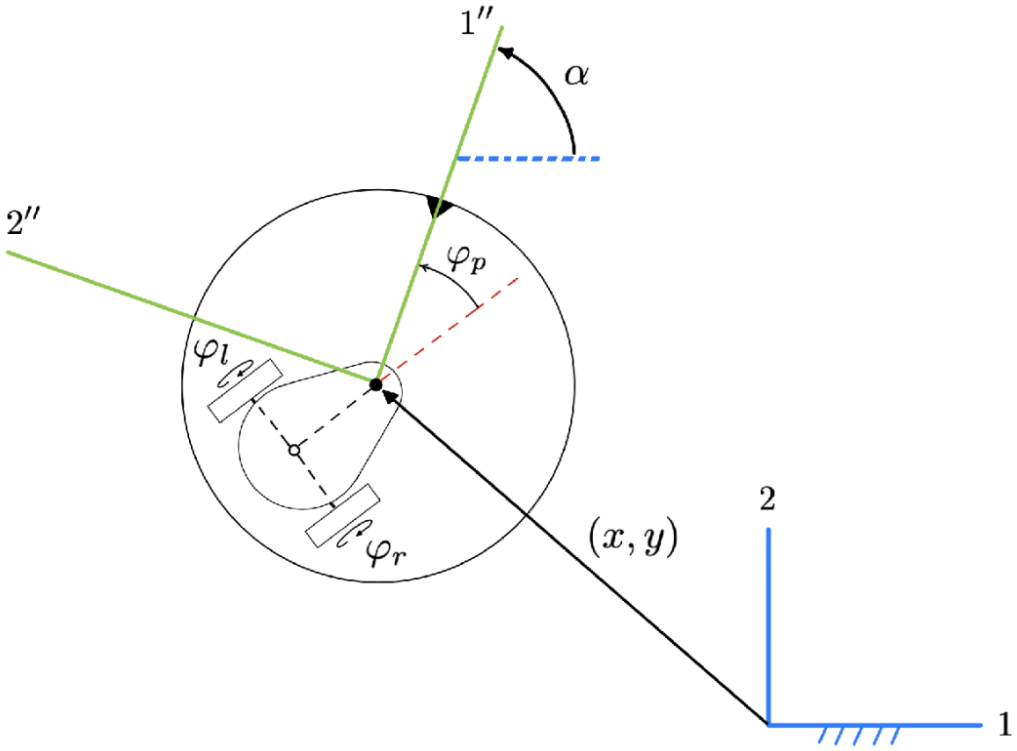


The blue frame is the absolute frame fixed to the ground. The red and green frames are fixed to the chassis and the platform, respectively. M is the midpoint of the wheels axis and P is the location of the pivot joint. The red axis 1' is always aligned with M and P. The angle between the green and red frames coincides with the pivot joint angle φ_p.

Each frame has a vector basis attached to it, whose unit vectors are directed along the frame axes. The three bases will be referred to as B = {1,2,3}, B' = {1', 2', 3'} and B'' = {1'', 2'', 3''} respectively.

Configuration and state coordinates

The robot configuration can be described by means of six coordinates: the absolute position (x, y) of the pivot joint, the absolute angle α of the platform, the pivot angle φ_p, and the angles of the right and left wheels, φ_r and φ_l, relative to the chassis.



The robot configuration is thus given by

$$\mathbf{q} = (x, y, \alpha, \varphi_r, \varphi_l, \varphi_p),$$

and the time derivative of $\mathbf{q}$ provides the robot velocity:

$$\dot{\mathbf{q}} = (\dot{x}, \dot{y}, \dot{\alpha}, \dot{\varphi_r}, \dot{\varphi_l}, \dot{\varphi_p})$$

Therefore, the robot state will be given by

$$\mathbf{x} = (\mathbf{q}, \dot{\mathbf{q}})$$

2. Kinematic constraints imposed by the rolling contacts

We next see that the coordinates of $\mathbf{x}$ are not independent. The rolling contacts of the wheels impose a kinematic constraint of the form

$$\mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} = \mathbf{0}$$

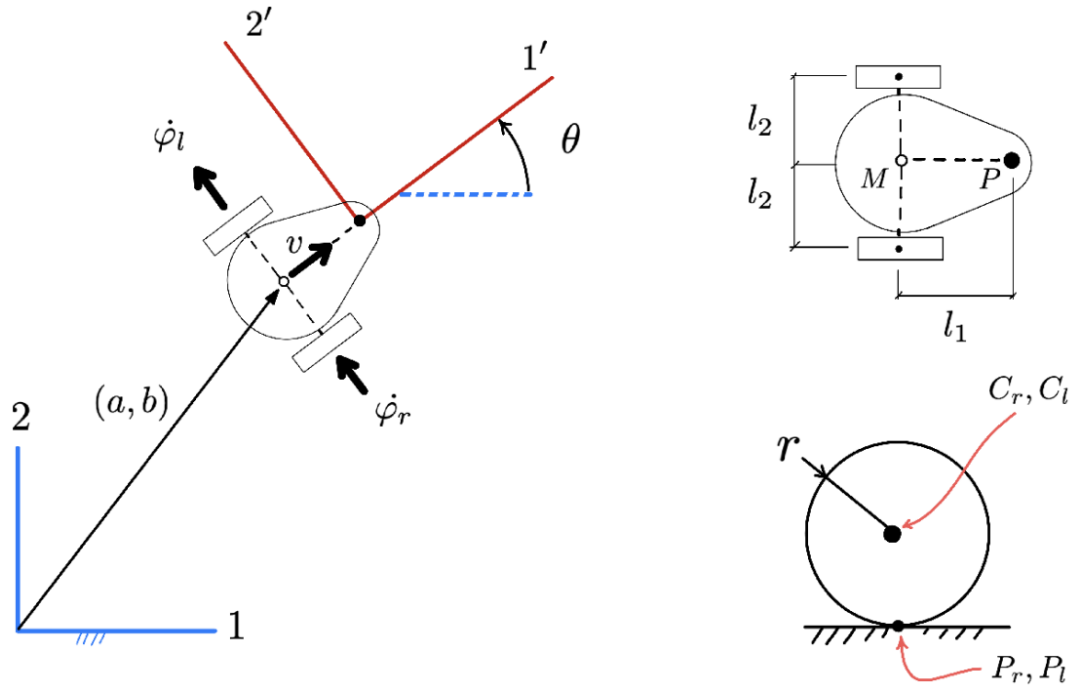
where $\mathbf{J}(\mathbf{q})$ is a 3×6 Jacobian matrix. We derive such a constraint in two steps:

1. We obtain the kinematic constraints imposed by the wheels.
2. We rewrite these constraints using the $\mathbf{x}$ variables only.

Let us see these steps in detail. On our derivations we use $\mathbf{v}(Q)$ to denote the velocity of a point Q of the robot. The basis in which $\mathbf{v}(Q)$ is expressed will be mentioned explicitly, or understood by context.

Kinematic constraints of the differential drive

For the moment we neglect the platform and focus on the chassis:



The chassis pose is given by the position vector (a, b) of point M, and by the orientation angle θ. M is the midpoint of the wheels axis, and P is the location of the pivot joint. We use l_1 and l_2 to refer to the pivot offset from M and the half-length of the wheels axis, respectively. Also, C_r and C_l denote the centers of the right and left wheels, and P_r and P_l are the contact points of such wheels with the ground. The two wheels have the same radius r.

The rolling contact constraints of the chassis can be found by computing the velocities of $\mathbf{v}(P_r)$ and $\mathbf{v}(P_l)$ in terms of $\dot{a}$, $\dot{b}$, $\dot{\theta}$, $\dot{\varphi}_r$, and $\dot{\varphi}_l$ (i.e., as if the robot were a floating kinematic tree) and forcing these velocities to be zero (as the wheels do not slide when placed on the ground).

To obtain $\mathbf{v}(P_r)$ and $\mathbf{v}(P_l)$, note first that the velocity of M can only be directed along axis 1', since lateral slipping is forbidden under perfect rolling. Therefore in B' = {1',2',3'}

$$\mathbf{v}(M) = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix}$$

Also note that,

$$\mathbf{v}(C_r) = \mathbf{v}(M) + \boldsymbol{\omega}_{\text{chassis}} \times \overrightarrow{MC_r} = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ -l_2 \\ 0 \end{bmatrix} = \begin{bmatrix} v + l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{v}(C_l) = \mathbf{v}(M) + \boldsymbol{\omega}_{\text{chassis}} \times \overrightarrow{MC_l} = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ l_2 \\ 0 \end{bmatrix} = \begin{bmatrix} v - l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix}$$

or, in Matlab syntax:

```
v_of_Cr = [v;0;0] + cross( [0;0;theta_dot], [0;-l_2;0] );  % Velocity of Cr
v_of_Cl = [v;0;0] + cross( [0;0;theta_dot], [0; l_2;0] );  % Velocity of Cl
```

The velocities of the ground contact points are thus given by

$$\mathbf{v}(P_r) = \mathbf{v}(C_r) + \boldsymbol{\omega}_{\text{wheel}} \times \overrightarrow{C_rP_r} = \begin{bmatrix} v + l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \dot{\varphi}_r \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -r \end{bmatrix} = \begin{bmatrix} v + l_2\,\dot{\theta} - r\,\dot{\varphi}_r \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{v}(P_l) = \mathbf{v}(C_l) + \boldsymbol{\omega}_{\text{wheel}} \times \overrightarrow{C_lP_l} = \begin{bmatrix} v - l_2\,\dot{\theta} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \dot{\varphi}_l \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -r \end{bmatrix} = \begin{bmatrix} v - l_2\,\dot{\theta} - r\,\dot{\varphi}_l \\ 0 \\ 0 \end{bmatrix}$$

i.e.,

```
v_of_Pr = v_of_Cr + cross( [0; varphi_dot_r; theta_dot], [0; 0; -r]);
v_of_Pl = v_of_Cl + cross( [0; varphi_dot_l; theta_dot], [0; 0; -r]);
```

Since P_r and P_l do not slip, $\mathbf{v}(P_r)$ and $\mathbf{v}(P_l)$ must be zero, which gives the two fundamental constraints of the robot:

```
eqn1 = (v_of_Pr(1) == 0);
```

```
eqn2 = (v_of_Pl(1) == 0);
disp(eqn1); disp(eqn2);
```

It is now easy to solve for v and $\dot{\theta}$ in these two equations:

```
[solved_v,solved_thetadot] = solve([eqn1,eqn2],[v,theta_dot]);

eqn3 = (v==simplify(solved_v));
eqn4 = (theta_dot==simplify(solved_thetadot));

disp(eqn3); disp(eqn4);
```

Note from the previous figure that in the basis B = {1,2,3}

$$\mathbf{v}(M) = \begin{bmatrix} \dot{a} \\ \dot{b} \\ 0 \end{bmatrix}$$

and that it must be

$$\begin{cases} \dot{a} = v \cdot \cos\theta \\ \dot{b} = v \cdot \sin\theta \end{cases}$$

By substituting $v = \frac{r\left(\dot{\varphi}_l + \dot{\varphi}_r\right)}{2}$ in these two equations, and also considering $\dot{\theta} = -\frac{r\left(\dot{\varphi}_l - \dot{\varphi}_r\right)}{2\,l_2}$, we obtain the system:

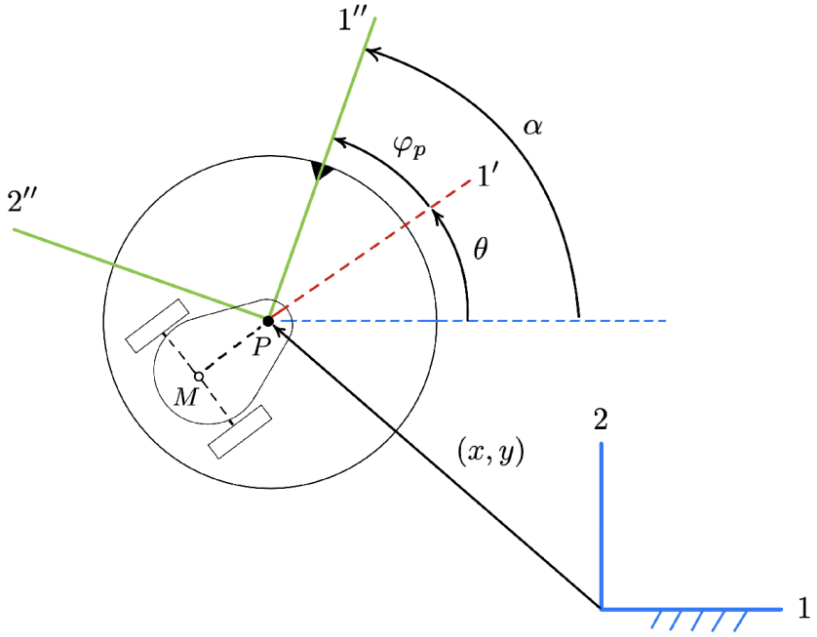
```
eqn5 = a_dot == solved_v * cos(theta);
eqn6 = b_dot == solved_v * sin(theta);
eqn7 = eqn4;

disp(eqn5); disp(eqn6); disp(eqn7);
```

This system provides the forward kinematic solution for the differential drive. It can also be viewed as the system of kinematic constraints that express the rolling contact constraints.

Kinematic constraints of the whole robot

To obtain the kinematic constraints of Otbot itself, we just need to rewrite the previous system using the variables in $\mathbf{x}$. This entails substituting θ, $\dot{\theta}$, $\dot{a}$, and $\dot{b}$ by their expressions in terms of the coordinates in $\mathbf{x}$.

By inspecting the figure



it is clear that

$$\theta = \alpha - \varphi_p$$

$$\dot{\theta} = \dot{\alpha} - \dot{\varphi}_p$$

and we also see that

$$\mathbf{v}(M) = \mathbf{v}(P) + \omega_{\text{chassis}} \times \overrightarrow{PM},$$

so using B = {1, 2, 3} we can write

$$\begin{bmatrix} \dot{a} \\ \dot{b} \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} \times \begin{bmatrix} -l_1 \cdot \cos\theta \\ -l_1 \cdot \sin\theta \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} + \dot{\theta}\, l_1 \sin\theta \\ \dot{y} - \dot{\theta}\, l_1 \cos\theta \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} + (\dot{\alpha} - \dot{\varphi}_p)\, l_1\, \sin(\alpha - \varphi_p) \\ \dot{y} - (\dot{\alpha} - \dot{\varphi}_p)\, l_1\, \cos(\alpha - \varphi_p) \\ 0 \end{bmatrix}$$

In sum we have the relationships

$$\begin{cases} \theta = \alpha - \varphi_p \\ \dot{\theta} = \dot{\alpha} - \dot{\varphi}_p \\ \dot{a} = \dot{x} + (\dot{\alpha} - \dot{\varphi}_p)\, l_1\, \sin(\alpha - \varphi_p) \\ \dot{b} = \dot{y} - (\dot{\alpha} - \dot{\varphi}_p)\, l_1\, \cos(\alpha - \varphi_p) \end{cases}$$

which give the desired values of θ, $\dot{\theta}$, $\dot{a}$, and $\dot{b}$ in terms of $\mathbf{q}$ and $\dot{\mathbf{q}}$. We thus can substitute these expressions in eqn5, eqn6, and eqn7 above to obtain eqn8, eqn9, and eqn10:

```
theta_value     = alpha - varphi_p;
theta_dot_value = alpha_dot - varphi_dot_p;
a_dot_value     = x_dot+(alpha_dot - varphi_dot_p)*l_1*sin(alpha-varphi_p);
b_dot_value     = y_dot-(alpha_dot - varphi_dot_p)*l_1*cos(alpha-varphi_p);

eqn8 = subs(eqn5,[a_dot,theta],[a_dot_value,theta_value]);
eqn9 = subs(eqn6,[b_dot,theta],[b_dot_value,theta_value]);
eqn10 = subs(eqn7,theta_dot,theta_dot_value);

disp(eqn8); disp(eqn9); disp(eqn10);
```

In matrix form, these equations can be written as

$$\mathbf{J}(\mathbf{q}) \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \\ \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix} = \mathbf{0},$$

where $\mathbf{J}(\mathbf{q})$ is the 3×6 matrix

```
J = equationsToMatrix(...
    [eqn8,eqn9,eqn10],...
    [x_dot,y_dot,alpha_dot,varphi_dot_r,varphi_dot_l,varphi_dot_p])
```

3. Solution to the instantaneous kinematic problems

Let $\mathbf{p} = (x, y, \alpha)$ and $\boldsymbol{\varphi} = (\varphi_r, \varphi_l, \varphi_p)$. These vectors provide the task and joint space variables of the robot. Their time derivatives $\dot{\boldsymbol{\varphi}}$ and $\dot{\mathbf{p}}$ are called the motor speeds and the platform twist, respectively

This section solves the following two problems:

- The forward instantaneous kinematic problem (**FIIKP**): Obtain $\dot{\mathbf{p}}$ as a function of $\dot{\boldsymbol{\varphi}}$.
- The inverse instantaneous kinematic problem (**IIKP**): Obtain $\dot{\boldsymbol{\varphi}}$ as a function of $\dot{\mathbf{p}}$.

It is also shown that the IIKP is always solvable, irrespective of the robot configuration. This implies that Otbot's platform can move omnidirectionally in the plane.

Forward problem

We only have to isolate $\dot{x}$, $\dot{y}$, and $\dot{\alpha}$ from the earlier equations eqn8, eqn9, and eqn10 defining $\mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} = \mathbf{0}$:

```
[solved_xdot, solved_ydot, solved_alphadot] = ...
    solve([eqn8,eqn9,eqn10],[x_dot,y_dot,alpha_dot]);

eqn11 = (x_dot == solved_xdot);
eqn12 = (y_dot == solved_ydot);
eqn13 = (alpha_dot == solved_alphadot);

disp(eqn11); disp(eqn12); disp(eqn13);
```

These equations directly provide $\dot{\mathbf{p}}$ as a function of $\dot{\boldsymbol{\varphi}}$. This function can be expressed as

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \end{bmatrix} = \mathbf{M}_{FIK}(\mathbf{q}) \cdot \begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix},$$

where $\mathbf{M}_{FIK}(\mathbf{q})$ is the following 3×3 matrix:

```
MFIK = equationsToMatrix([solved_xdot, solved_ydot, solved_alphadot],...
                         [varphi_dot_r, varphi_dot_l, varphi_dot_p])
```

Inverse problem

We now wish to find $\dot{\boldsymbol{\varphi}}$ as a function of $\dot{\mathbf{p}}$. Clearly, this function is given by

$$\begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix} = \mathbf{M}_{IIK}(\mathbf{q}) \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \end{bmatrix},$$

where $\mathbf{M}_{IIK}(\mathbf{q}) = \mathbf{M}_{FIK}^{-1}(\mathbf{q})$.

Note that the determinant of $\mathbf{M}_{FIK}(\mathbf{q})$ is

```
detMFIK = simplify(det(MFIK))
```

detMFIK =

$$-\frac{l_1 r^2}{2 l_2}$$

Since r, l_1, and l_2 are all positive in Otbot, we see that $\mathbf{M}_{FIK}^{-1}(\mathbf{q})$ always exists, so the IIKP is solvable in all configurations of the robot. An important consequence is that the platform will be able to move under any twist $\dot{\mathbf{p}}$ in the plane. In other words, it will be an omnidirectional platform.

$\mathbf{M}_{IIK}(\mathbf{q})$ has the expression:

```
MIIK = simplify(inv(MFIK))
```

4. The kinematic model in control form

These equations

$$\begin{cases} \dot{\mathbf{p}} = \mathbf{M}_{FIK}(\mathbf{q}) \cdot \dot{\boldsymbol{\varphi}} \\ \dot{\boldsymbol{\varphi}} = \dot{\boldsymbol{\varphi}} \end{cases}$$

can be written in the usual form used in control engineering,

$$\dot{\mathbf{q}} = \mathbf{f}_{\text{kin}}(\mathbf{q}, \mathbf{v}),$$

where $\mathbf{v} = \dot{\boldsymbol{\varphi}}$ and

$$\mathbf{f}_{\text{kin}}(\mathbf{q}, \mathbf{v}) = \begin{bmatrix} \mathbf{M}_{\text{FIK}}(\mathbf{q}) \\ \mathbf{I}_3 \end{bmatrix} \cdot \dot{\boldsymbol{\varphi}}.$$

Notice that in this model $\mathbf{v}$ plays the role of the control actions, which are motor velocities in this case.

We could use this model for trajectory planning already. In doing so, we would neglect the system dynamics, but a strong point is the model simplicity, which leads to faster planners and controllers. Moreover, the model would be sufficient if the commanded velocities $\mathbf{v}(t)$ where easy to control (e.g., if $\mathbf{v}(t)$ is smooth enough for the motors at hand). The model would also be helpful to perform early tests of a planner under development, or to compute approximate trajectories to be later enhanced by an optimizer (one using the full dynamical model for example).

5. Save matrices

```
save('MIIK.mat',"MIIK")
save('MFIK.mat',"MFIK")
save('J.mat',"J")
```

6. Print ellapsed time

```
toc
```

Dynamic model of Otbot

This live script develops the dynamic model of Otbot and the solution to its forward and inverse dynamics.

Table of Contents

Recall that the equation of motion of a robot takes the form

$$\mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\,\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}(\mathbf{q})^{\top}\boldsymbol{\lambda} = \mathbf{Q}_a(\mathbf{u}) + \mathbf{Q}_f$$

where:

- $\mathbf{q}$ is the configuration vector of the robot (of size $n_q = 6$ in our case)
- $\mathbf{u} = [\boldsymbol{\tau}_r, \boldsymbol{\tau}_l, \boldsymbol{\tau}_p]^{\top}$ is the vector of motor torques (the right and left wheel torques and the platform torque).
- $\mathbf{M}(\mathbf{q})$ is the mass matrix of the unconstrained system (positive-definite of size $n_q \times n_q$)
- $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$ is the generalized Coriolis and centrifugal force matrix
- $\mathbf{G}(\mathbf{q})$ is the generalized gravity force
- $\mathbf{J}(\mathbf{q})$ is the constraint Jacobian of the robot
- $\boldsymbol{\lambda}$ is a vector of Lagrange mulitpliers.
- $\mathbf{Q}_a(\mathbf{u})$ is the generalized force of actuation, which can be written in the form $\mathbf{E}(\mathbf{q})\cdot\mathbf{u}$
- $\mathbf{Q}_f$ is the generalised force modelling all friction forces in the system

Note that, since the Otbot will move on flat terrain, $\mathbf{G}(\mathbf{q}) = \mathbf{0}$. Friction forces will also be neglected for the moment, so $\mathbf{Q}_f$ will be zero initially. The constraint Jacobian $\mathbf{J}(\mathbf{q})$ was already obtained in kinematic_model.mlx and we will simply load it from an appropriate mat file.

Our task thus boils down to obtaining $\mathbf{M}(\mathbf{q})$, $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$, and $\mathbf{E}(\mathbf{q})$ initially. After that, we will assemble the full dynamic model using the acceleration-level constraint (the time derivative of $\mathbf{J}\cdot\dot{\mathbf{q}} = \mathbf{0}$).

Hereafter, when we say "chassis" we mean the robot chassis **including** the wheels.

1. Initializations

```
% Clear all variables, close all figures, and clear the command window
clearvars
close all
clc

% Start stopwatch
tic;

% Display the matrices with rectangular brackets
sympref('MatrixWithSquareBrackets',true);

% Avoid Matlab's own substitution of long expressions
sympref('AbbreviateOutput',false);

% Symbolic variables to be used (see the figures and explanations below)
syms x y             % Absolute coords of the pivot joint
syms x_dot y_dot     % Absolute velocity components of the pivot joint
syms alpha           % Absolute angle of the platform
syms alpha_dot       % Absolute angular velocity of the platform
syms varphi_r        % Angle of right wheel relative to chassis
syms varphi_l        % Angle of left wheel relative to chassis
syms varphi_p        % Pivot joint angle (platform wrt chassis)
syms varphi_dot_l    % Angular velocity of left wheel
syms varphi_dot_r    % Angular velocity of right wheel
syms varphi_dot_p    % Angular velocity of the platform relative to the chassis
syms l_1             % Pivot offset relative to the wheels axis
syms l_2             % One half of the wheels separation
syms m_c             % Mass of the chassis (including wheels)
syms m_p             % Mass of the platform
syms x_F y_F         % Coords of F (c.o.m. of the platform) in platform frame
syms x_B y_B         % Coords of B (c.o.m. of the chassis) in chassis frame
syms I_p             % Vertical moment of inertia of the platform at its c.o.m.
syms I_a             % Axial moment of inertia of one wheel
syms I_c             % Vertical moment of inertia of the chassis at its c.o.m.
```

2. Computation of the mass matrix

Recall that the robot configuration is given by

$$\mathbf{q} = (x, y, \alpha, \varphi_r, \varphi_l, \varphi_p)$$

To find the mass matrix $\mathbf{M}(\mathbf{q})$ we first write the kinetic energy T of the robot as a function of $\mathbf{q}$ and $\dot{\mathbf{q}}$ and then express it as:

$$T(\mathbf{q}, \dot{\mathbf{q}}) = \frac{1}{2} \dot{\mathbf{q}}^{\top} \cdot \mathbf{M}(\mathbf{q}) \cdot \dot{\mathbf{q}}$$

We have to compute the translational and rotational kinetic energies of all bodies in the Otbot, and add them all to obtain T.

```
% Chassis angle theta, and its derivative, in terms of qdot
theta = alpha - varphi_p;
theta_dot = alpha_dot - varphi_dot_p;

% Translational kinetic energies

    % Of the chassis
    T_tra_c = kin_en_trans(x_B, y_B, m_c, x_dot, y_dot, theta, theta_dot);

    % Of the platform
    T_tra_p = kin_en_trans(x_F, y_F, m_p, x_dot, y_dot, alpha, alpha_dot);

% Rotational kinetic energies

    % Whole chassis including wheels
    T_rot_c = 1/2 * I_c * (theta_dot)^2;

    % Right wheel when turning about its axis alone
    T_rot_r = 1/2 * I_a * (varphi_dot_r)^2;

    % Left wheel when turing about its axis alone
    T_rot_l = 1/2 * I_a * (varphi_dot_l)^2;

    % Platform
    T_rot_p = 1/2 * I_p * (alpha_dot)^2;

% Total kinetic energy
T = T_tra_c + T_tra_p + ...
    T_rot_c + T_rot_r + T_rot_l + T_rot_p;

% Display the results
display(T_tra_c); display(T_tra_p);
display(T_rot_c); display(T_rot_r); display(T_rot_l); display(T_rot_p);
display(T);
```

Since T is a quadratic form, the Hessian of T gives the desired mass matrix:

```
M = hessian(T,[x_dot y_dot alpha_dot varphi_dot_r varphi_dot_l varphi_dot_p])
```

3. Computation of the Coriolis matrix

Recall that the (i, j) element of $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ is given by the following formula (see Murray, Sastry, Lee)

$$\mathbf{C}_{ij} = \frac{1}{2}\sum_{k=1}^{n}\left(\frac{\partial \mathbf{M}_{ij}}{\partial \mathbf{q}_k} + \frac{\partial \mathbf{M}_{ik}}{\partial \mathbf{q}_j} - \frac{\partial \mathbf{M}_{kj}}{\partial \mathbf{q}_i}\right)\dot{\mathbf{q}}_k,$$

which in Matlab can be implemented as follows:

```
qvec = [x y alpha varphi_r varphi_l varphi_p].';
qdotvec = [x_dot y_dot alpha_dot varphi_dot_r varphi_dot_l varphi_dot_p].';

nq = length(qvec);
C = sym(zeros(nq,nq));

for i=1:nq
    for j=1:nq
        for k=1:nq
            C(i,j) = C(i,j) + qdotvec(k) * 0.5 * ( ...
                diff(M(i,j),qvec(k)) + ...
                diff(M(i,k),qvec(j)) - ...
                diff(M(k,j),qvec(i))                    );
        end
    end
end
C = simplify(C);
display(C)
```

4. Generalized force of actuation

Our vector $\mathbf{u}$ of motor torques is defined as

$$\mathbf{u} = \begin{bmatrix} \tau_r \\ \tau_l \\ \tau_p \end{bmatrix}$$

```
syms tau_r tau_l tau_p
uvec = [tau_r; tau_l; tau_p];
```

Each of the torques in $\mathbf{u}$ acts directly on a $\dot{\mathbf{q}}_i$ coordinate and, therefore, the generalized force of actuation is given by

$$\mathbf{Q}_a = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \tau_r \\ \tau_l \\ \tau_p \end{bmatrix}$$

To rewrite this force in the form $\mathbf{Q}_a = \mathbf{E} \cdot \mathbf{u}$ we only have to define

$$\mathbf{E}(\mathbf{q}) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
Ematrix = [zeros(3);eye(3)];
```

This completes our derivation of all matrices involved in the equation of motion.

5. Acceleration constraint

We now wish to obtain the robot model in the form $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u})$. This requires writing $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ as a function of $\mathbf{q}$, $\dot{\mathbf{q}}$, and $\mathbf{u}$. Recall that the Euler-Lagrange equation is

$$\mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}} + \mathbf{J}^\top \boldsymbol{\lambda} = \mathbf{E}\,\mathbf{u}$$ (we omit the dependencies on $\mathbf{q}$ and $\dot{\mathbf{q}}$ for simplicity)

Since this equation is a system of 6 equations in 9 unknowns (6 coordinates in $\ddot{\mathbf{q}}$ and 3 in $\boldsymbol{\lambda}$) we need 3 additional equations to determine $\ddot{\mathbf{q}}$ and $\boldsymbol{\lambda}$ for a given $\mathbf{u}$. These equations can be obtained by taking the time derivative of the kinematic constraint

$$\mathbf{J} \cdot \dot{\mathbf{q}} = \mathbf{0},$$

which gives

$$\mathbf{J}\,\ddot{\mathbf{q}} + \dot{\mathbf{J}}\,\dot{\mathbf{q}} = \mathbf{0},$$

or, equivalently,

$$\mathbf{J}\,\ddot{\mathbf{q}} = -\dot{\mathbf{J}}\,\dot{\mathbf{q}}$$

The latter equation is called the acceleration constraint of the robot.

Recall that $\mathbf{J}$ was obtained in `kinematic_model.mlx`. Therefore, we only need to calculate its time derivative now:

```
% We first import J (obtained by an earlier run of kinematic_model.mlx)
load("J.mat")

% Substitute its variables by functions of t and take the time derivative
syms f1(t) f2(t)
J_of_t = subs(J,[alpha varphi_p],[f1(t) f2(t)]);
dJdt = diff(J_of_t,t);

% Substitute d/dt of f1 and f2 by the original variables using dot notation
Jdot = subs(dJdt,[f1(t) f2(t) diff(f1,t) diff(f2,t)], ...
    [alpha varphi_p alpha_dot varphi_dot_p])
```

6. Final model in explicit first-order form

The equations

$$\begin{cases} \mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}} + \mathbf{J}^{\top}\,\boldsymbol{\lambda} = \mathbf{E}\,\mathbf{u} \\ \mathbf{J}\,\ddot{\mathbf{q}} = -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{cases}$$

can be written as a linear system with the form

$$\begin{bmatrix} \mathbf{M} & \mathbf{J}^{\top} \\ \mathbf{J} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{E}\,\mathbf{u} - \mathbf{C}\,\dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{bmatrix}$$

and since $\mathbf{M}$ is positive-definite and $\mathbf{J}$ is full row rank, the matrix on the left-hand side can be inverted to write

$$\begin{bmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}^{\top} \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} - \mathbf{C}\,\dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{bmatrix}.$$

Therefore

$$\ddot{\mathbf{q}} = \begin{bmatrix} \mathbf{I}_6 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{M} & \mathbf{J}^{\top} \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} - \mathbf{C}\,\dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{bmatrix}.$$

Finally, by considering the trivial equation $\dot{\mathbf{q}} = \dot{\mathbf{q}}$ in conjunction with the earlier equation we arrive at

$$\begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}} \\ \begin{bmatrix} \mathbf{I}_6 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{M} & \mathbf{J}^{\top} \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} - \mathbf{C}\,\dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{bmatrix} \end{bmatrix},$$

which is the robot model in the usual control form $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$.

7. Solution to the forward dynamics problem

The forward dynamics problem consists in finding the acceleration $\ddot{\mathbf{q}}$ that corresponds to a given $\mathbf{u} = [\tau_r, \tau_l, \tau_p]^\top$. Such a $\ddot{\mathbf{q}}$ is given by the earlier expression

$$\ddot{\mathbf{q}} = \begin{bmatrix} \mathbf{I}_6 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{M} & \mathbf{J}^\top \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} - \mathbf{C}\,\dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{bmatrix}.$$

8. Solution to the inverse dynamics problem

The inverse dynamics problem consists in finding the torques $\mathbf{u} = [\tau_r, \tau_l, \tau_p]^\top$ that produce a desired $\ddot{\mathbf{q}}$. These torques are obtained by solving

$$\mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}} + \mathbf{J}^\top \boldsymbol{\lambda} = \mathbf{E}\,\mathbf{u}$$

for $\mathbf{u}$ and $\boldsymbol{\lambda}$ (6 equations and 6 unknowns). For this we define $\boldsymbol{\tau}_{\mathbf{ID}} = \mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}}$ and rewrite the previous equation as

$$\mathbf{E}\,\mathbf{u} - \mathbf{J}^\top \boldsymbol{\lambda} = \boldsymbol{\tau}_{\mathbf{ID}},$$

or, equivalently, as

$$\begin{bmatrix} \mathbf{E} & -\mathbf{J}^\top \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \boldsymbol{\tau}_{\mathbf{ID}}.$$

It is easy to see that $\begin{bmatrix} \mathbf{E} & -\mathbf{J}^\top \end{bmatrix}$ is a 6×6 full rank matrix irrespectively of $\mathbf{q}$. This follows directly from the expressions of $\mathbf{E}$ and $\mathbf{J}$ in the Otbot. Thus, we can write

$$\begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{E} & -\mathbf{J}^\top \end{bmatrix}^{-1} \boldsymbol{\tau}_{\mathbf{ID}},$$

so that

$$\mathbf{u} = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{E} & -\mathbf{J}^\top \end{bmatrix}^{-1} \boldsymbol{\tau}_{\mathbf{ID}}$$

provides the desired value for $\mathbf{u}$.

9. Save wokspace and main matrices to file

```
save('M.mat',"M")
save('C.mat',"C")
```

10. Print ellapsed time

```
toc
```

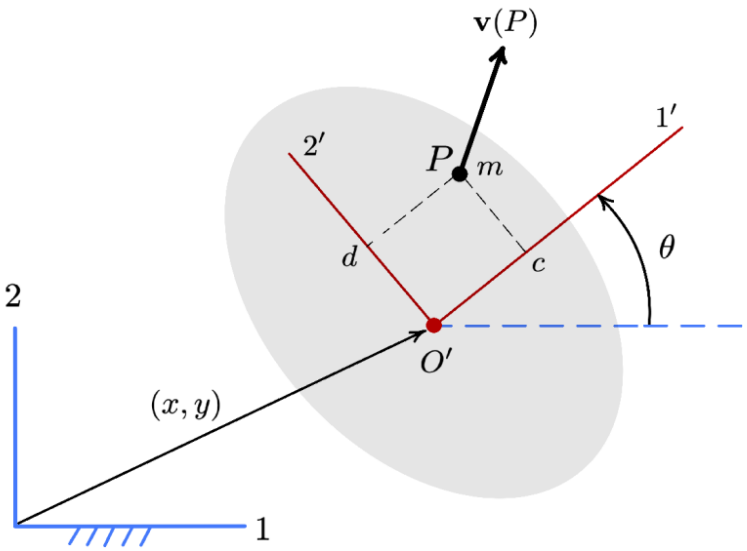
```
Elapsed time is 8.576265 seconds.
```

Appendix: Function kin_en_trans

This function computes the translational kinetic energy of a point mass located at P on the grey body, in terms of the velocity coordinates $\dot{x}$, $\dot{y}$, and $\dot{\theta}$ of the body.

The following notation is used:

- (c, d) = Local coordinates of P in the body-fixed frame {1',2'}
- (x, y) = Absolute coordinates of the origin of the body-fixed frame
- θ = absolute angle of the body
- m = mass of P



This function can be used to compute the translational kinetic energy of any body in planar motion. Just let P be the c.o.m. of the body, and m be the total mass of the body.

```
function T = kin_en_trans(c,d,m,xdot,ydot,theta,thetadot)

    cth = cos(theta);
    sth = sin(theta);

    A = [1,    0,    -cth * d - sth * c;
         0,    1,     cth * c - sth * d;
         0,    0,              1          ];

    Mp = [m, 0, 0; 0, m, 0; 0, 0, 0];
    M = simplify(transpose(A) * Mp * A);
    w = [xdot;ydot;thetadot];
    T = expand(1/2 * transpose(w) * M * w,'ArithmeticOnly',true);

end
```

Dynamic model in task space coordinates

This live script obtains a closed formula for the inverse dynamics of the Otbot by using a tricky elimination of the Lagrange multipliers. It then obtains the equation of motion in task-space coordinates, which can be used to design a computed-torque controller for trajectory tracking.

Table of Contents

1 - Initializations

```
% Clear all variables, close all figures, and clear the command window
clearvars
close all
clc

% Start stopwatch
tic;

% Display the matrices with rectangular brackets
sympref('MatrixWithSquareBrackets',true);

% Avoid Matlab's own substitution of long expressions
sympref('AbbreviateOutput',false);

% Symbolic variables to be used (see the figures and explanations below)
syms x y             % Absolute coords of the pivot joint
syms x_dot y_dot     % Absolute velocity components of the pivot joint
syms alpha           % Absolute angle of the platform
syms alpha_dot       % Absolute angular velocity of the platform
syms varphi_r        % Angle of right wheel
syms varphi_l        % Angle of left wheel
syms varphi_p        % Pivot joint angle
syms varphi_dot_l    % Angular velocity of left wheel
syms varphi_dot_r    % Angular velocity of right wheel
syms varphi_dot_p    % Angular velocity of the pivot motor
syms l_1             % Pivot offset relative to the wheels axis
syms l_2             % One half of the wheels separation
syms m_c             % Mass of the chassis including the wheels
syms m_p             % Mass of the platform
syms x_B y_B         % Coords of the c.o.m. of the chassis in chassis frame
syms x_F y_F         % Coords of the c.o.m. of the platform in platform frame
```

```
syms I_c              % Vertical moment of inertia of the chassis at B
syms I_p              % Vertical moment of inertia of the platform at F
syms I_a              % Axial moment of inertia of one wheel

% Load matrices
load('MIIK.mat')
load('MFIK.mat')
load('M.mat')
load('C.mat')
```

2 - Multiplier-free inverse dynamics

Our goal is to obtain an equation of motion that describes the time evolution of the $\mathbf{p}$ coordinates alone, and at the same time does not contain the annoying Lagrange multipliers $\boldsymbol{\lambda}$. In this way we will obtain a one-to-one relationship between platform accelerations and the torques $\mathbf{u}$ applied to the robot. This relationship can later be used to design a computed-torque control law.

Consider the following parametrizations of the feasible $\dot{\mathbf{q}}$

$$\dot{\mathbf{q}} = \boldsymbol{\Lambda} \cdot \dot{\mathbf{p}},$$

$$\dot{\mathbf{q}} = \boldsymbol{\Delta} \cdot \dot{\boldsymbol{\varphi}},$$

where

$$\boldsymbol{\Lambda} = \begin{bmatrix} \mathbf{I}_3 \\ \mathbf{M}_{\mathrm{IIK}} \end{bmatrix}$$

$$\boldsymbol{\Delta} = \begin{bmatrix} \mathbf{M}_{\mathrm{FIK}} \\ \mathbf{I}_3 \end{bmatrix}$$

```
Lambda = [eye(3); MIIK]
Delta  = [MFIK; eye(3)]
```

For later use, also consider the time derivative of the first parameterization

$$\ddot{\mathbf{q}} = \boldsymbol{\Lambda}\ddot{\mathbf{p}} + \dot{\boldsymbol{\Lambda}}\dot{\mathbf{p}}.$$

and let us compute $\dot{\boldsymbol{\Lambda}}$ with Matlab:

```
% Substitute its variables by functions of t
syms f1(t) f2(t)
Lambda2 = subs(Lambda,[alpha varphi_p],[f1(t) f2(t)]);
% Take the time derivative
dLambdadt = diff(Lambda2,t);

% Substitute d/dt of f1 and f2 by the original variables using dot notation
```

```
Lambda_dot = subs(dLambdadt,...
    [f1(t) f2(t) diff(f1,t) diff(f2,t)], ...
    [alpha varphi_p alpha_dot varphi_dot_p])

clearvars f1(t) f2(t)
```

Recall that the equation of motion of the Otbot takes the form

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{C}\dot{\mathbf{q}} + \mathbf{J}^\top\boldsymbol{\lambda} = \mathbf{E}\,\mathbf{u}$$

Let us multiply this equation by $\boldsymbol{\Delta}^\top$:

$$\boldsymbol{\Delta}^\top\mathbf{M}\ddot{\mathbf{q}} + \boldsymbol{\Delta}^\top\mathbf{C}\dot{\mathbf{q}} + \boldsymbol{\Delta}^\top\mathbf{J}^\top\boldsymbol{\lambda} = \boldsymbol{\Delta}^\top\mathbf{E}\mathbf{u}$$

Note that $\boldsymbol{\Delta}^\top\mathbf{J}^\top\boldsymbol{\lambda} = \mathbf{0}$, as the columns of $\boldsymbol{\Delta}$ form a basis of the kernel of $\mathbf{J}$, and $\mathbf{J}^\top\boldsymbol{\lambda}$ is a vector orthogonal to this kernel. Also note that

$$\boldsymbol{\Delta}^\top\mathbf{E} = \begin{bmatrix}\mathbf{M}_{\mathrm{FIK}}^\top & \mathbf{I}_3\end{bmatrix}\begin{bmatrix}\mathbf{0}\\ \mathbf{I}_3\end{bmatrix} = \mathbf{I}_3,$$

so the equation of motion reduces to

$$\boldsymbol{\Delta}^\top\mathbf{M}\ddot{\mathbf{q}} + \boldsymbol{\Delta}^\top\mathbf{C}\dot{\mathbf{q}} = \mathbf{u}$$

This formula gives us an explicit solution for the inverse dynamics of the Otbot.

3 - Equation of motion in task-space coordinates

If we now substitute $\dot{\mathbf{q}} = \boldsymbol{\Lambda}\dot{\mathbf{p}}$ and $\ddot{\mathbf{q}} = \boldsymbol{\Lambda}\ddot{\mathbf{p}} + \dot{\boldsymbol{\Lambda}}\dot{\mathbf{p}}$ into the earlier equation we obtain:

$$\boldsymbol{\Delta}^\top\mathbf{M}\boldsymbol{\Lambda}\ddot{\mathbf{p}} + \boldsymbol{\Delta}^\top\left(\mathbf{M}\dot{\boldsymbol{\Lambda}} + \mathbf{C}\boldsymbol{\Lambda}\right)\dot{\mathbf{p}} = \mathbf{u}$$

Let us compute

$$\bar{\mathbf{M}} = \boldsymbol{\Delta}^\top\mathbf{M}\boldsymbol{\Lambda}$$

$$\bar{\mathbf{C}} = \boldsymbol{\Delta}^\top\left(\mathbf{M}\dot{\boldsymbol{\Lambda}} + \mathbf{C}\boldsymbol{\Lambda}\right)$$

```
M_bar = simplify(Delta.' * M * Lambda)
C_bar = simplify(Delta.' * (M * Lambda_dot + C * Lambda) )
```

We call these matrices the task-space mass matrix, and the task-space Coriolis matrix respectively. Using them, the equation of motion can be written in the compact form

$$\bar{\mathbf{M}}\,\ddot{\mathbf{p}} + \bar{\mathbf{C}}\,\dot{\mathbf{p}} = \mathbf{u}$$

This is called the equation of motion in task-space coordinates, and can be used to design a computed-torque controller to track arbitrary trajectories in task space.

4 - Save main matrices

```
save('Lambda.mat',"Lambda")
save('Lambda_dot.mat',"Lambda_dot")
save('Delta.mat',"Lambda")
save('M_bar.mat',"M_bar")
save('C_bar.mat',"C_bar")
```

5 - Print ellapsed time

```
toc
```

A.2 Dynamic simulation routines

This section shows the code used in dynamic simulation. The scheme of all the programs used is quite simple and described in Fig. A.1. We start by loading the `.mat` files generated by the *liveScripts* of the Section A.1. Then we execute the script `otbot_parameters.m` which we configure the dynamical parameters of the model together with other basic elements that we will need, such as rotation matrices or geometric dimensions of the robot. The results of this script are stored in a file named `m_struc.mat`. Then we use the script called `write_matrices.m`, which uses the dynamical parameters of the previous file and also all the matrices and symbolic vectors we had obtained earlier. With this, we convert the symbolic matrices and vectors into simple MATLAB functions that depend only on the system state. This script will create a file called `sm_struc.mat` containing all of these new functions. After this, it is required to run the script `Main_Simulation.m` if we desire to simulate the dynamic model. This script configures the parameters of the simulation such as the duration and the integration routine to be used, the variables to be plot, whether to activate the control law, the initial conditions, whether we want friction in the model or not, whether we want to have force disturbances throughout the simulation and whether we wish a video to be generated at the end of the simulation. This main script uses the data and functions stored in the `m_struc.mat` and `sm_struc.mat` files to be able to work properly. As the script executes, it calls several other functions. One of the most important is the one that implements the dynamic model in explicit first-order form, named `xdot_otbot.m`.

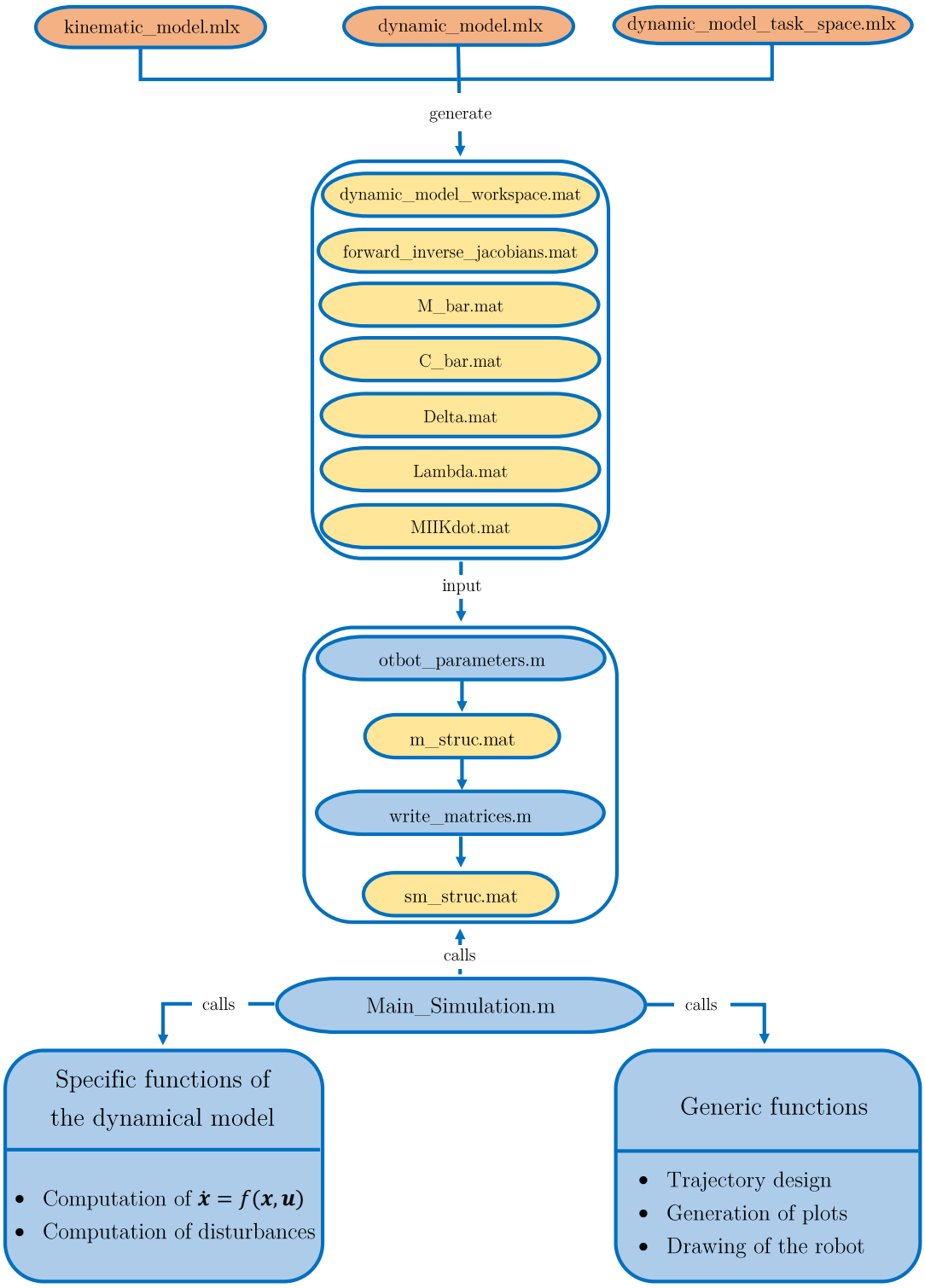


Figure A.1: Organization chart that describes the structure of the code for dynamic simulation.

A.2.1 Otbot parameters

```
%% Setting up the physical parameters of the robot

%———————— Inertia parameters ————————%

% Central moment of inertia of the chassis body about axis 3 [kg*m2]
m.I_b = 1.06458;

% Central moment of inertia of the platform body about axis 3'' [kg*m2]
m.I_p = 2.22223;

% Axial moment of inertia of one wheel [kg*m2]
m.I_a = 1.03570*1e-2;

% Twisting moment of inertia of one wheel [kg*m2]
m.I_t = 5.61007*1e-3;

%———————— Geometric parameters ————————%

% Pivot offset relative to the wheels axis [m]
m.l_1 = 0.25;

% One half of the wheels separation [m]
m.l_2 = 0.20;

% Wheel radius [m]
m.r = 0.10;

%———————— Mass parameters ————————%

% Mass of the chassis base [kg]
m.m_b = 105.00;

% Mass of one wheel [kg]
m.m_w = 2.0714;

% Mass of the platform [kg]
m.m_p = 21.94795;

%———————— Centers of mass coordinates ————————%

% x coord of the c.o.m. of the chassis body in the chassis frame [m]
m.x_G = -m.l_1/2;
```

```
% y coord of the c.o.m. of the chassis body in the chassis frame [m]
m.y_G = 0;


% x coord of the c.o.m. of the platform body in the platform frame [m]
m.x_F = 0;

% y coord of the c.o.m. of the platform body in the platform frame [m]
m.y_F = 0;

%———————— Dynamic friction coefficient ————————%

% Viscous friction coefficient [kg*m2*s-1]
m.b_frict = [0.18;0.18;0.18];

%———————— Parameters to Draw ————————%

% Matrix to define the body in relative frame with the pivot point at the
% origin
m.CBprel = [0, -m.l_1, -m.l_1;
          0, -m.l_2, +m.l_2;
          zeros(1,3)];

% Matrix to define the body in relative frame with the pivot point at the
% origin
m.RWBprel = [ + m.r,        + m.r,        - m.r,        - m.r ;
           + (m.l_2)/6, - (m.l_2)/6,  - (m.l_2)/6, + (m.l_2)/6;
           zeros(1,4)];

% Generic rotation matrix Rz

syms ang

Rzmat = [cos(ang), -sin(ang), 0;
         sin(ang), cos(ang),  0;
          0,          0,      1];

m.Rzmat = matlabFunction(Rzmat);


%% Saving m structure
Upath = userpath;
savedirsp1 = strcat(Upath,'\Sim_Otbot\Dyn_Sim\Otbot1_CTC_W1_V2_FDPFD\');
save(strcat(savedirsp1,'m_struc.mat'),'m')
clearvars
```

```
close all
clc

%% Run the other script in order to update matrices

write_matrices
```

A.2.2 Write matrices

```
load('m_struc')
load('dynamic_model_workspace')
load('M_bar.mat')
load('C_bar.mat')
load('forward_inverse_jacobians.mat')
load('Lambda.mat')
load('Delta.mat')
load('MIIKdot.mat')

%% Create missing simbolic variables

syms r

%% Seting up

% Now we will be seting up this matrices in a apropiate way in order to use
% them for the simulations.

% First of all C_bar2
C_barmatrix2 = subs(C_bar_V2,[I_b, I_p, I_a, I_t, l_1, l_2, m_b, m_w, m_p, ...
    x_G, y_G, x_F, y_F, r], [m.I_b, m.I_p, m.I_a, m.I_t, m.l_1, m.l_2, ...
    m.m_b, m.m_w, m.m_p, m.x_G, m.y_G, m.x_F, m.y_F, m.r]);

% Save matrices in a structure array
sm.C_barmatrix2 = matlabFunction(C_barmatrix2);

% Then M_bar2
M_barmatrix2 = subs(M_bar_V2,[I_b, I_p, I_a, I_t, l_1, l_2, m_b, m_w, m_p, ...
    x_G, y_G, x_F, y_F, r], [m.I_b, m.I_p, m.I_a, m.I_t, m.l_1, m.l_2, ...
    m.m_b, m.m_w, m.m_p, m.x_G, m.y_G, m.x_F, m.y_F, m.r]);

% Save matrices in a structure array
sm.M_barmatrix2 = matlabFunction(M_barmatrix2);

% MIIK matrix
MIIKmatrix = subs(MIIK_of_q,[l_1,l_2,r],[m.l_1,m.l_2,m.r]);
```

```
% Save matrices in a structure array
sm.MIIKmatrix = matlabFunction(MIIKmatrix);

% MIIKdot matrix
MIIKdotmatrix = subs(MIIKdot,[l_1,l_2,r],[m.l_1,m.l_2,m.r]);

% Save matrices in a structure array
sm.MIIKdotmatrix = matlabFunction(MIIKdotmatrix);

% MFIK matrix
MFIKmatrix = subs(MFIK_of_q,[l_1,l_2,r],[m.l_1,m.l_2,m.r]);

% Save matrices in a structure array
sm.MFIKmatrix = matlabFunction(MFIKmatrix);

% Lambda Matrix
Lambdamatrix = subs(Lambda,[l_1,l_2,r],[m.l_1,m.l_2,m.r]);

% Save matrix in a structure array
sm.Lambdamatrix = matlabFunction(Lambdamatrix);

% Lambda Matrix
Deltamatrix = subs(Delta,[l_1,l_2,r],[m.l_1,m.l_2,m.r]);

% Save matrix in a structure array
sm.Deltamatrix = matlabFunction(Deltamatrix);


%% Add the experssion of the Kinetic Energy

Texpr = subs(T,[I_b, I_p, I_a, I_t, l_1, l_2, m_b, m_w, m_p, x_G, y_G,...
    x_F, y_F, r], [m.I_b, m.I_p, m.I_a, m.I_t, m.l_1, m.l_2, m.m_b, ...
    m.m_w, m.m_p, m.x_G, m.y_G, m.x_F, m.y_F, m.r]);

% Save kinetic energy expression in structure array
sm.Texpr = matlabFunction(Texpr);

%% Saveing only the structure object and deleting everything else
Upath = userpath;
savedirsp1 = strcat(Upath,'\Sim_Otbot\Dyn_Sim\Otbot1_CTC_W1_V2_FDPFD\');
save(strcat(savedirsp1,'sm_struc.mat'),'sm')
clearvars
close all
clc
```

A.2.3 Main simulation

```
clearvars
close all
clc
%% Flags
% Flag to define if you want video output or not
FlagVideo = 'YES';
               % (FlagVideo == YES)> simulation with VIDEO output
               % (FlagVideo == NO )> simulation without VIDEO output

% Flag to define if you want a constnat torque for all simulation or not
u_Flag = 'CTE';
               % (u_Flag == CTE)> simulation with constant torques
               % (u_Flag == VAR)> simulation with a torque function of time

                % Now we create torques comand vector
                u = [0.1;0.1;-0.1]; % All in Nm

% Flag to define if we want to add friction in our model or not
FrictFlag = 'NO';
               % (FrictFlag == YES)> Our model will have frictions
               % (FrictFlag == NO)> Our model will not contain frictions

% Flag to define if we want to add non modelled disturbances during the
% simulation
DistFlag = 'NO';
               % (DistFlag == YES)> Non modelled disturbances will be added
               % (DistFlag == NO)> Non modelled disturbances will NOT be added

% Flag to define if you want to track the path of each center of mass
TrackFlag = 'PF';
               % (TrackFlag == NO)> plots without any paths of center of
               % mass plotted

               % (TrackFlag == CB)> plots the path of the center of mass of
               % the chassis body

               % (TrackFlag == PF)> plots the path of the center of mass of
               % the platform body

               % (TrackFlag == BOTH)> plots the path of both centers of
               % mass

% Flag to choose if you want to compute and display the plots of action vector u
```

```
UvecFlag = 'YES';
                % (UvecFlag == YES)> action vector u plots will be displayed
                % (UvecFlag == NO)> action vector u plots will not be displayed

KinEnFlag = 'NO'; % Flag to choose if Kinetic energic plot must be displayed or
                  % not
               % (KinEnFlag == YES)> Outputs a plot of kinetic energy
               % (KinEnFlag == NO )> No output plot of kinetic energy

Jdot_qdotFlag = 'NO'; % Flag to choose if plot of equation Jdot*qdot is equal
                      % to 0 during all simulation
               % (Jdot_qdotFlag == YES)> Outputs a plot of this equation
               % (Jdot_qdotFlag == NO)> No output of this plot
```

```
%% Setting up simulation parameters

tf = 15;      % Final simulation time
h = 0.001;    % Number of samples within final vectors (times and states)
opts = odeset('MaxStep',1e-3); % Max integration time step
```

```
%% Setting up model parameters & matrices
load('m_struc')
load('sm_struc')
```

```
%% Initial conditions
p0 = [0,0,0]';
varphi0 = [0,0,0]';

pdot0 = zeros(3,1);

% Need MIIK
MIIKmat = sm.MIIKmatrix(p0(3),varphi0(3));
% Computing p0
varphidot0 = MIIKmat*pdot0;

% Initial conditions for the system
xs0=[p0; varphi0; pdot0; varphidot0];
```

```
%% Building the diferential equation
dxdt= @(t,xs) xdot_otbot(t, xs, m, sm, u, u_Flag, ...
    FrictFlag, DistFlag);
```

```
%% Simulationg with ode45
t = 0:h:tf;
[times,states]=ode45(dxdt,t,xs0,opts);
```

```
%% Vectorize the path of centers of mass
if strcmp(TrackFlag,'NO')==0

    % Setting this to adjust it to the video, here and down below they must
    % heve the same values, n & fs
    fs=30;
    n=round(1/(fs*h));

    % Set, ploting factor:
    % This plot factor is to scale if we want compute the com of each body every
    % frame (Gpf = 1) or less frequently i.e every 2 frames (Gpf = 2...)
    Gpf = 16;
    n2 = Gpf*n;

    aux1=length(times);
    GBpmat = zeros(3,round(aux1/n2));
    GPpmat = zeros(3,round(aux1/n2));
    jaux=1;

    for i = 1:n2:aux1
        q.x = states(i,1);
        q.y = states(i,2);
        q.alpha = states(i,3);
        q.varphi_r = states(i,4);
        q.varphi_l = states(i,5);
        q.varphi_p = states(i,6);

        [GBpi,GPpi] = c_o_m_bodies(m,q);
        GBpmat(:,jaux) = GBpi;
        GPpmat(:,jaux) = GPpi;
        jaux=jaux+1;
    end
end
```

```
%% Compute KinEnergy in Every instant
if strcmp(KinEnFlag,'YES')
    ls = length(states);
    kinenvalues = zeros(ls,1);
   for i=1:ls
       kinenvalues(i,1) = sm.Texpr(states(i,3),states(i,9),states(i,6), ...
           states(i,11),states(i,12),states(i,10),states(i,7),states(i,8));
   end
end
```

```
%% Compute action torques in every instant
if strcmp(UvecFlag,'YES')
    u_vector = zeros(tf/h+1,4);
    switch u_Flag
        case 'CTE'
            ls = length(states);
            for i=1:ls
                u_vector(i,1:4) = [u',times(i)];
            end
        case 'VAR'
            ls = length(states);
            for i=1:ls
                u_f = u_function(times(i),u);
                u_vector(i,1:4) = [u_f',times(i)];
            end
        otherwise
            disp('Warning this u_Flag does not exits omiting torques' ...
                'camputation')
            disp('Computation with CTE torques will be launched')
            ls = length(states);
            for i=1:ls
                u_vector(i,1:4) = [u',times(i)];
            end
    end
else
    disp('Plots of the evolution of actions u will not be displayed')
end
```

```
%% Plot results
figure;
plot(times,states(:,1))
xlabel('t(s)','Interpreter','latex'),ylabel('$x$(meters)','Interpreter', ...
    'latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,2))
xlabel('t(s)','Interpreter','latex'),ylabel('$y$(meters)','Interpreter', ...
    'latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,3))
xlabel('t(s)','Interpreter','latex'),ylabel('$\alpha$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
```

```
plot(times,states(:,4))
xlabel('t(s)','Interpreter','latex'),ylabel('$\varphi_r$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,5))
xlabel('t(s)','Interpreter','latex'),ylabel('$\varphi_l$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,6))
xlabel('t(s)','Interpreter','latex'),ylabel('$\varphi_p$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,7))
xlabel('t(s)','Interpreter','latex'),ylabel('$\dot{x}$(meters/second)', ...
    'Interpreter','latex'),title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,8))
xlabel('t(s)','Interpreter','latex'),ylabel('$\dot{y}$(meters/second)', ...
    'Interpreter','latex'),title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,9))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\alpha}$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,10))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\varphi}_r$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,11))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\varphi}_l$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,12))
xlabel('t(s)','Interpreter','latex'), ...
```

```
ylabel('$\dot{\varphi}_p$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

% Plot the action vectors
if strcmp(UvecFlag,'YES')
    figure;
    plot(times,u_vector(:,1:2))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\tau$ of wheels(N$\cdot$m)','Interpreter','latex'), ...
    title('Action Torques','Interpreter','latex')
    legend('$\tau$ right','$\tau$ left','Interpreter','latex')

    figure;
    plot(times,u_vector(:,3))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\tau$ pivot(N$\cdot$m)','Interpreter','latex'), ...
    title('Action Torques','Interpreter','latex')
end

if strcmp(KinEnFlag,'YES')
    figure;
    plot(times,kinenvalues)
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('Kinetic Energy (Joules)','Interpreter','latex'), ...
    title('System Energy','Interpreter','latex')
end
```

```
%% Seting Jdot_qdot = 0 Flag

if strcmp(Jdot_qdotFlag,'YES')
    ls = length(states);
    J_qvalues = zeros(ls,3);
    for i=1:ls
        Jplot = sm.Jmatrix(states(i,3),states(i,6));
        qdotplot = states(i,7:12)';
        J_qvalues(i,:) = (Jplot*qdotplot)';
    end
    figure;
    plot(times,J_qvalues)
    xlabel('t(s)'),ylabel('Result vector'), title('Vector result of J*qdot')
end
```

```
%% Movie and video of the simulation
switch FlagVideo
    case 'YES'
```

```
switch TrackFlag
    case 'NO'
        time=cputime;
        % Animation
        % h is the sampling time
        % n is the scaling factor in order not to plot with the same step
        % than during the integration with ode45

        fs=30;
        n=round(1/(fs*h));

        % Set up the movie.
        writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
        %writerObj.FileFormat = 'mp4';
        writerObj.FrameRate = fs; % How many frames per second.
        open(writerObj);

        TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
        if strcmp(TargetFlag,'NO') || TEFlagcheck
            for i = 1:n:length(times)
                    q.x = states(i,1);
                    q.y = states(i,2);
                    q.alpha = states(i,3);
                    q.varphi_r = states(i,4);
                    q.varphi_l = states(i,5);
                    q.varphi_p = states(i,6);

                    elapsed = cputime-time;
                    if elapsed > 1200
                        disp(elapsed);
                        disp('took too long to generate the video')
                        break
                    else
                        draw_otbot(m,q) % Drawing frame
                        hold on
                        if strcmp(DistFlag,'YES') && ...
                                norm(Dist_funct(times(i))) ~= 0
                            Dist_video = Dist_funct(times(i));
                            D_XY = Dist_video(1:2,1);
                            D_XY = D_XY./norm(D_XY);
                            D_XY = 3*m.l_1*D_XY;
                            PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                                D_XY(1,1)));
                            arrows(states(i,1) - D_XY(1,1), ...
                                states(i,2) - D_XY(2,1), norm(D_XY), ...
```

```
                        270 - PolAng , 'FaceColor','r', ...
                        'EdgeColor','none');
                end
                hold off

                % 'gcf' can handle if you zoom in to take a movie
                    .
                frame = getframe(gcf);
                writeVideo(writerObj, frame);
            end

        end
        close(writerObj); % Saves the movie.

    elseif strcmp(TargetFlag,'YES') && TEFlag == 0
        for i = 1:n:length(times)
                q.x = states(i,1);
                q.y = states(i,2);
                q.alpha = states(i,3);
                q.varphi_r = states(i,4);
                q.varphi_l = states(i,5);
                q.varphi_p = states(i,6);

                elapsed = cputime-time;
                if elapsed > 1200
                    disp(elapsed);
                    disp('took too long to generate the video')
                    break
                else
                    draw_otbot(m,q)
                    hold on
                    otbot_circle_v2(targetpos(i,1), ...
                        targetpos(i,2),m.l_2/4,'r');
                    hold off

                    hold on
                    if strcmp(DistFlag,'YES') && ...
                            norm(Dist_funct(times(i))) ~= 0
                        Dist_video = Dist_funct(times(i));
                        D_XY = Dist_video(1:2,1);
                        D_XY = D_XY./norm(D_XY);
                        D_XY = 3*m.l_1*D_XY;
                        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                            D_XY(1,1)));
                        arrows(states(i,1) - D_XY(1,1), ...
```

```
                                states(i,2) - D_XY(2,1), norm(D_XY), ...
                                270 - PolAng , 'FaceColor','r', ...
                                'EdgeColor','none');
                        end
                        hold off

                        % 'gcf' can handle if you zoom in to take a movie
                            .
                        frame = getframe(gcf);
                        writeVideo(writerObj, frame);
                    end

                end
                close(writerObj); % Saves the movie.

            end

        case 'BOTH'
            time=cputime;
            % Animation
            % h is the sampling time
            % n is the scaling factor in order not to plot with the same step
            % than during the integration with ode45

            fs=30;
            n=round(1/(fs*h));

            % Set up the movie.
            writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
            %writerObj.FileFormat = 'mp4';
            writerObj.FrameRate = fs; % How many frames per second.
            open(writerObj);
            jaux = 1;
            jaux2 = 1;

            TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
            if strcmp(TargetFlag,'NO') || TEFlagcheck
                for i = 1:n:length(times)
                        q.x = states(i,1);
                        q.y = states(i,2);
                        q.alpha = states(i,3);
                        q.varphi_r = states(i,4);
                        q.varphi_l = states(i,5);
                        q.varphi_p = states(i,6);
```

```
elapsed = cputime-time;
if elapsed > 1200
    disp(elapsed);
    disp('took too long to generate the video')
    break
else
    draw_otbot(m,q)
    if jaux2 == 1
        GBpplot = GBpmat(:,1:jaux);
        GPpplot = GPpmat(:,1:jaux);
        multi_circles(GBpplot,m.l_2/6,'b')
        hold on
        multi_circles(GPpplot,m.l_2/6,'g')
        jaux2 = jaux2 + 1;
        jaux = jaux+1;
    elseif jaux2>= Gpf
        multi_circles(GBpplot,m.l_2/6,'b')
        hold on
        multi_circles(GPpplot,m.l_2/6,'g')
        jaux2 = 1;
    else
        multi_circles(GBpplot,m.l_2/6,'b')
        hold on
        multi_circles(GPpplot,m.l_2/6,'g')
        jaux2 = jaux2 + 1;
    end

    hold on
    if strcmp(DistFlag,'YES') && ...
            norm(Dist_funct(times(i))) ~= 0
        Dist_video = Dist_funct(times(i));
        D_XY = Dist_video(1:2,1);
        D_XY = D_XY./norm(D_XY);
        D_XY = 3*m.l_1*D_XY;
        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
            D_XY(1,1)));
        arrows(states(i,1) - D_XY(1,1), ...
            states(i,2) - D_XY(2,1), norm(D_XY), ...
            270 - PolAng , 'FaceColor','r', ...
            'EdgeColor','none');
    end
    hold off

    % 'gcf' can handle if you zoom in to take a movie
        .
```

```
                frame = getframe(gcf);
                writeVideo(writerObj, frame);
            end
        end
        close(writerObj); % Saves the movie.

    elseif strcmp(TargetFlag,'YES') && TEFlag == 0
        for i = 1:n:length(times)
            q.x = states(i,1);
            q.y = states(i,2);
            q.alpha = states(i,3);
            q.varphi_r = states(i,4);
            q.varphi_l = states(i,5);
            q.varphi_p = states(i,6);

            elapsed = cputime-time;
            if elapsed > 1200
                disp(elapsed);
                disp('took too long to generate the video')
                break
            else
                draw_otbot(m,q)
                if jaux2 == 1
                    GBpplot = GBpmat(:,1:jaux);
                    GPpplot = GPpmat(:,1:jaux);
                    multi_circles(GBpplot,m.l_2/6,'b')
                    hold on
                    multi_circles(GPpplot,m.l_2/6,'g')
                    hold on
                    otbot_circle_v2(targetpos(i,1), ...
                        targetpos(i,2),m.l_2/4,'r');
                    hold off
                    jaux2 = jaux2 + 1;
                    jaux = jaux+1;
                elseif jaux2>= Gpf
                    multi_circles(GBpplot,m.l_2/6,'b')
                    hold on
                    multi_circles(GPpplot,m.l_2/6,'g')
                    hold on
                    otbot_circle_v2(targetpos(i,1), ...
                        targetpos(i,2),m.l_2/4,'r');
                    hold off
                    jaux2 = 1;
                else
                    multi_circles(GBpplot,m.l_2/6,'b')
```

```
                            hold on
                            multi_circles(GPpplot,m.l_2/6,'g')
                            hold on
                            otbot_circle_v2(targetpos(i,1), ...
                                targetpos(i,2),m.l_2/4,'r');
                            hold off
                            jaux2 = jaux2 + 1;
                        end

                        hold on
                        if strcmp(DistFlag,'YES') && ...
                                norm(Dist_funct(times(i))) ~= 0
                            Dist_video = Dist_funct(times(i));
                            D_XY = Dist_video(1:2,1);
                            D_XY = D_XY./norm(D_XY);
                            D_XY = 3*m.l_1*D_XY;
                            PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                                D_XY(1,1)));
                            arrows(states(i,1) - D_XY(1,1), ...
                                states(i,2) - D_XY(2,1), norm(D_XY), ...
                                270 - PolAng , 'FaceColor','r', ...
                                'EdgeColor','none');
                        end
                        hold off

                        % 'gcf' can handle if you zoom in to take a movie
                            .
                        frame = getframe(gcf);
                        writeVideo(writerObj, frame);
                    end
                end
                close(writerObj); % Saves the movie.

            end

        case 'CB'
            time=cputime;
            % Animation
            % h is the sampling time
            % n is the scaling factor in order not to plot with the same step
            % than during the integration with ode45

            fs=30;
            n=round(1/(fs*h));
```

```
% Set up the movie.
writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
%writerObj.FileFormat = 'mp4';
writerObj.FrameRate = fs; % How many frames per second.
open(writerObj);
jaux = 1;
jaux2 = 1;

TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
if strcmp(TargetFlag,'NO') || TEFlagcheck
    for i = 1:n:length(times)
            q.x = states(i,1);
            q.y = states(i,2);
            q.alpha = states(i,3);
            q.varphi_r = states(i,4);
            q.varphi_l = states(i,5);
            q.varphi_p = states(i,6);

            elapsed = cputime-time;
            if elapsed > 1200
                disp(elapsed);
                disp('took too long to generate the video')
                break
            else
                draw_otbot(m,q)
                if jaux2 == 1
                    GBpplot = GBpmat(:,1:jaux);
                    multi_circles(GBpplot,m.l_2/6,'b')
                    jaux2 = jaux2 + 1;
                    jaux = jaux+1;
                elseif jaux2>= Gpf
                    multi_circles(GBpplot,m.l_2/6,'b')
                    jaux2 = 1;
                else
                    multi_circles(GBpplot,m.l_2/6,'b')
                    jaux2 = jaux2 + 1;
                end

                hold on
                if strcmp(DistFlag,'YES') && ...
                        norm(Dist_funct(times(i))) ~= 0
                    Dist_video = Dist_funct(times(i));
                    D_XY = Dist_video(1:2,1);
                    D_XY = D_XY./norm(D_XY);
                    D_XY = 3*m.l_1*D_XY;
```

```
                        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                            D_XY(1,1)));
                        arrows(states(i,1) - D_XY(1,1), ...
                            states(i,2) - D_XY(2,1), norm(D_XY), ...
                            270 - PolAng , 'FaceColor','r', ...
                            'EdgeColor','none');
                    end
                    hold off

                    % 'gcf' can handle if you zoom in to take a movie
                        .
                    frame = getframe(gcf);
                    writeVideo(writerObj, frame);
                end
            end
            close(writerObj); % Saves the movie.

        elseif strcmp(TargetFlag,'YES') && TEFlag == 0
            for i = 1:n:length(times)
                    q.x = states(i,1);
                    q.y = states(i,2);
                    q.alpha = states(i,3);
                    q.varphi_r = states(i,4);
                    q.varphi_l = states(i,5);
                    q.varphi_p = states(i,6);

                    elapsed = cputime-time;
                    if elapsed > 1200
                        disp(elapsed);
                        disp('took too long to generate the video')
                        break
                    else
                        draw_otbot(m,q)
                        if jaux2 == 1
                            GBpplot = GBpmat(:,1:jaux);
                            multi_circles(GBpplot,m.l_2/6,'b')
                            hold on
                            otbot_circle_v2(targetpos(i,1), ...
                                targetpos(i,2),m.l_2/4,'r');
                            hold off
                            jaux2 = jaux2 + 1;
                            jaux = jaux+1;
                        elseif jaux2>= Gpf
                            multi_circles(GBpplot,m.l_2/6,'b')
                            hold on
```

```
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = 1;
                    else
                        multi_circles(GBpplot,m.l_2/6,'b')
                        hold on
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = jaux2 + 1;
                    end

                    hold on
                    if strcmp(DistFlag,'YES') && ...
                            norm(Dist_funct(times(i))) ~= 0
                        Dist_video = Dist_funct(times(i));
                        D_XY = Dist_video(1:2,1);
                        D_XY = D_XY./norm(D_XY);
                        D_XY = 3*m.l_1*D_XY;
                        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                            D_XY(1,1)));
                        arrows(states(i,1) - D_XY(1,1), ...
                            states(i,2) - D_XY(2,1), norm(D_XY), ...
                            270 - PolAng , 'FaceColor','r', ...
                            'EdgeColor','none');
                    end
                    hold off

                    % 'gcf' can handle if you zoom in to take a movie
                        .
                    frame = getframe(gcf);
                    writeVideo(writerObj, frame);
                end
            end
            close(writerObj); % Saves the movie.
        end

    case 'PF'
        time=cputime;
        % Animation
        % h is the sampling time
        % n is the scaling factor in order not to plot with the same step
        % than during the integration with ode45
```

```
fs=30;
n=round(1/(fs*h));

% Set up the movie.
writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
%writerObj.FileFormat = 'mp4';
writerObj.FrameRate = fs; % How many frames per second.
open(writerObj);
jaux = 1;
jaux2 = 1;

TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
if strcmp(TargetFlag,'NO') || TEFlagcheck
    for i = 1:n:length(times)
            q.x = states(i,1);
            q.y = states(i,2);
            q.alpha = states(i,3);
            q.varphi_r = states(i,4);
            q.varphi_l = states(i,5);
            q.varphi_p = states(i,6);

            elapsed = cputime-time;
            if elapsed > 1200
                disp(elapsed);
                disp('took too long to generate the video')
                break
            else
                draw_otbot(m,q)
                if jaux2 == 1

                    GPpplot = GPpmat(:,1:jaux);

                    multi_circles(GPpplot,m.l_2/6,'g')
                    jaux2 = jaux2 + 1;
                    jaux = jaux+1;
                elseif jaux2>= Gpf

                    multi_circles(GPpplot,m.l_2/6,'g')
                    jaux2 = 1;
                else

                    multi_circles(GPpplot,m.l_2/6,'g')
                    jaux2 = jaux2 + 1;
                end
```

```
                    hold on
                    if strcmp(DistFlag,'YES') && ...
                            norm(Dist_funct(times(i))) ~= 0
                        Dist_video = Dist_funct(times(i));
                        D_XY = Dist_video(1:2,1);
                        D_XY = D_XY./norm(D_XY);
                        D_XY = 3*m.l_1*D_XY;
                        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                            D_XY(1,1)));
                        arrows(states(i,1) - D_XY(1,1), ...
                            states(i,2) - D_XY(2,1), norm(D_XY), ...
                            270 - PolAng , 'FaceColor','r', ...
                            'EdgeColor','none');
                    end
                    hold off

                    % 'gcf' can handle if you zoom in to take a movie
                        .
                    frame = getframe(gcf);
                    writeVideo(writerObj, frame);
                end
            end
            close(writerObj); % Saves the movie.

        elseif strcmp(TargetFlag,'YES') && TEFlag == 0
            for i = 1:n:length(times)
                    q.x = states(i,1);
                    q.y = states(i,2);
                    q.alpha = states(i,3);
                    q.varphi_r = states(i,4);
                    q.varphi_l = states(i,5);
                    q.varphi_p = states(i,6);

                    elapsed = cputime-time;
                    if elapsed > 1200
                        disp(elapsed);
                        disp('took too long to generate the video')
                        break
                    else
                        draw_otbot(m,q)
                        if jaux2 == 1
                            GPpplot = GPpmat(:,1:jaux);
                            multi_circles(GPpplot,m.l_2/6,'g')
                            hold on
                            otbot_circle_v2(targetpos(i,1), ...
```

```
                        targetpos(i,2),m.l_2/4,'r');
                    hold off
                    jaux2 = jaux2 + 1;
                    jaux = jaux+1;
                elseif jaux2>= Gpf
                    multi_circles(GPpplot,m.l_2/6,'g')
                    hold on
                    otbot_circle_v2(targetpos(i,1), ...
                        targetpos(i,2),m.l_2/4,'r');
                    hold off
                    jaux2 = 1;
                else
                    multi_circles(GPpplot,m.l_2/6,'g')
                    hold on
                    otbot_circle_v2(targetpos(i,1), ...
                        targetpos(i,2),m.l_2/4,'r');
                    hold off
                    jaux2 = jaux2 + 1;
                end

                hold on
                if strcmp(DistFlag,'YES') && ...
                        norm(Dist_funct(times(i))) ~= 0
                    Dist_video = Dist_funct(times(i));
                    D_XY = Dist_video(1:2,1);
                    D_XY = D_XY./norm(D_XY);
                    D_XY = 3*m.l_1*D_XY;
                    PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                        D_XY(1,1)));
                    arrows(states(i,1) - D_XY(1,1), ...
                        states(i,2) - D_XY(2,1), norm(D_XY), ...
                        270 - PolAng , 'FaceColor','r', ...
                        'EdgeColor','none');
                end
                hold off

                % 'gcf' can handle if you zoom in to take a movie
                    .
                frame = getframe(gcf);
                writeVideo(writerObj, frame);
            end
        end
        close(writerObj); % Saves the movie.
    end
```

```
            otherwise
                disp('This TrackFlag does not exist, omitting the video')
        end
    case 'NO'
        disp('Simulation without video launched');
    otherwise
        disp('This FlagVideo input does not exist');
end
```

A.2.4 Dynamic equations

```
function [ xdot ] = xdot_otbot(t, xs, m, sm, u, u_Flag, ...
    FrictFlag, DistFlag)
%   Differential equation (non-linearized), that modelizes our Otbot
%   system

% x = xs(1);
% y = xs(2);
alpha = xs(3);
% varphi_r = xs(4);
% varphi_l = xs(5);
varphi_p = xs(6);

% x_dot = xs(7);
% y_dot = xs(8);
alpha_dot = xs(9);
varphi_dot_r = xs(10);
varphi_dot_l = xs(11);
varphi_dot_p = xs(12);

% Setting fricctions
switch FrictFlag
    case 'YES'
        tau_friction(1:3,1) = zeros(3,1);
        tau_friction(4,1) = -m.b_frict(1,1)*varphi_dot_r; % tau_friction_right
        tau_friction(5,1) = -m.b_frict(2,1)*varphi_dot_l; % tau_friction_left
        tau_friction(6,1) = -m.b_frict(3,1)*varphi_dot_p; % tau_friction_pivot
    case 'NO'
        tau_friction = zeros(6,1);
    otherwise
        disp('Warning: This FrictFlag does not exist. System will be' ...
            'simulated without friction')
        tau_friction = zeros(6,1);
end
```

```
% Setting Distrurbances
switch DistFlag
    case 'YES'
        D_vec = Dist_funct(t);
    case 'NO'
        D_vec = zeros(6,1);
    otherwise
        disp('Warning: This DistFlag does not exist. System will be' ...
            'simulated without disturbances')
        D_vec = zeros(6,1);
end

MIIKs = sm.MIIKmatrix(alpha,varphi_p);
M_bars2 = sm.M_barmatrix2(alpha,varphi_p);
C_bars2 = sm.C_barmatrix2(alpha,alpha_dot,varphi_p,varphi_dot_p);
MIIKdots = sm.MIIKdotmatrix(alpha,alpha_dot,varphi_p,varphi_dot_p);
Dmats = sm.Deltamatrix(alpha,varphi_p);

Msys2 = [M_bars2, zeros(3);
         -MIIKs, eye(3)];

qdot = xs(7:12);

switch u_Flag
    case 'CTE'
        qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction + ...
            Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

    case 'VAR'
        u_f= u_function(t,u);
        qdotdot = pinv(Msys2)*[u_f + Dmats.'*tau_friction + ...
            Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

    otherwise
        disp('WARNING THIS FLAG DOES NOT EXIST (xdot_otbot)')
        disp('Simulation with CTE torques will be launched')
        qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction +  ...
            Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

end
xdot=[qdot; qdotdot];
end
```

A.2.5 Input function of time

```
function [ u_f ] = u_function( t, u )
%U_FUNCTION_LAGRANGEV2 Summary of this function goes here
%   Detailed explanation goes here

if t>1
    u_f = [0;0;0];
else
    u_f = u;
end

end
```

A.2.6 Disturbances function of time

```
function [D] = Dist_funct(t)
%DIST_FUNCT Summary of this function goes here
%   Detailed explanation goes here

if t>5 && t<6
    D = zeros(6,1);
    D(1:2,1) = [150;0];
elseif t>10 && t<11
    D = zeros(6,1);
    D(1:2,1) = [0;150];
elseif t>15 && t<16
    D = zeros(6,1);
    D(1:2,1) = [150;0];
elseif t>20 && t<21
    D = zeros(6,1);
    D(1:2,1) = [0;-150];
else
    D = zeros(6,1);
end

end
```

A.2.7 Compute center of mass coordinates

```
function [GBp,GPp] = c_o_m_bodies(m,q)
%C_O_M_BODIES Compute the position of centers of mass
%   This function is designed to calculate the centers of mass of the
%   chassis body and the platform body

xs(1)=q.x;
xs(2)=q.y;
xs(3)=q.alpha;
```

```
xs(4)=q.varphi_r;
xs(5)=q.varphi_l;
xs(6)=q.varphi_p;

% Defining rotation matrix
Rmat = m.Rzmat(xs(3)-xs(6));

Rmat2 = m.Rzmat(xs(3));

trvec = [xs(1);xs(2);0];

% Center of mass of the Chassis body
GBp = Rmat*[m.x_G; m.y_G; 0] + trvec;

% Center of mass of the platform
GPp = Rmat2*[m.x_F; m.y_F; 0] + trvec;

end
```

A.2.8 Drawing functions

```
function [  ] = draw_otbot( m, q )
%draw_otbot Summary of this function goes here
%   Function designed to draw the robot in a particular configuration
figure(100);
pos_fig1 = [0 0 1920 1080];
set(gcf,'Position',pos_fig1)

xs(1)=q.x;
xs(2)=q.y;
xs(3)=q.alpha;
xs(4)=q.varphi_r;
xs(5)=q.varphi_l;
xs(6)=q.varphi_p;

% Preparing workspace
clf(figure(100))
hold on;
grid on;
box on;
axis equal

%%%%%%%%%%%%%%%%%
view([0,90]); % this part sets the view of the plot coment if you want isometric
```

```
%%%%%%%%%%%%%%%%%%
axis([-3 3 -3 3]);

xticks(-100:1:100);
yticks(-100:1:100);

% Defining rotation matrix
Rmat = m.Rzmat(xs(3)-xs(6));

Rmat2 = m.Rzmat(xs(3));

% Drawing chassis Body

trvec = [xs(1);xs(2);0];

CBp = Rmat*m.CBprel + [trvec,trvec,trvec]; % Rotation + Translation

fill3(CBp(1,:),CBp(2,:),CBp(3,:),(1/255)*[191,191,191]);

% Drawing Wheels
RWBp_r = Rmat*m.RWBprel; % Rotation

RWBp_l = RWBp_r;   % Rotation is the same for both wheels

RWBp_r = RWBp_r + [CBp(:,2),CBp(:,2),CBp(:,2),CBp(:,2)]; % Translation
RWBp_l = RWBp_l + [CBp(:,3),CBp(:,3),CBp(:,3),CBp(:,3)]; % Translation

% Drawing the right wheel
fill3(RWBp_r(1,:),RWBp_r(2,:),RWBp_r(3,:),(1/255)*[68,68,68]);
% Drawing the left wheel
fill3(RWBp_l(1,:),RWBp_l(2,:),RWBp_l(3,:),(1/255)*[68,68,68]);

% Plot the platform

otbot_circle_v2(xs(1),xs(2),0.45,'NO');

% Drawing the orientation line
plot3([xs(1),0.45*cos(xs(3))+xs(1)],[xs(2),0.45*sin(xs(3))+xs(2)],[0,0], ...
    'color','r')

trih = 0.45/6;
triedge = trih/cos(pi/6);
tricords0 = [0,             trih  ,              trih;
             0, triedge*sin(pi/6) , -triedge*sin(pi/6);
             zeros(1,3)]; % Triangle coords at the origin
```

```
trirot = Rmat2*tricords0; % Rotate triangle
tricords = trirot + [(0.45-trih)*cos(xs(3))+xs(1);(0.45-trih)*sin(xs(3)) + ...
    xs(2);0];
fill(tricords(1,:),tricords(2,:),'r','EdgeColor','none') % Drawing

% Plotting center of mass of the chassis body
GBp = Rmat*[m.x_G; m.y_G; 0] + trvec;

otbot_circle_v2(GBp(1),GBp(2),m.l_2/6,'b');

% Plotting center of mass of the platform
GPp = Rmat2*[m.x_F; m.y_F; 0] + trvec;

otbot_circle_v2(GPp(1),GPp(2),m.l_2/6,'g');


hold off;
end

function varargout=arrows(x,y,l,az,varargin)
%ARROWS  Generalized 2-D arrows plot
%       ARROWS(X,Y,L,AZ) or ARROWS(X,Y,L,AZ,'Polar') draws an arrow on the
%       current axis at position X,Y with length L and azimuth AZ (in degrees,
%       clockwise from positive Y-axis direction).
%
%       ARROWS(X,Y,U,V,'Cartesian') uses arrows cartesian components U,V
%       instead of length/azimuth. This is an equivalent of QUIVER(X,Y,U,V,0).
%
%       ARROWS(X,Y,R,AZ,'Loop') draws a clockwise 3/4 loop arrow of radius R
%       and offset azimuth angle AZ (in degrees). Use negative value of R for a
%       counter clockwise loop.
%
%       X and Y can be scalars or matrix. In the last case, ARROWS will draw as
%       many arrows as elements of X and Y. Any or both pairs of parameters
%       L/AZ, U/V or R/AZ can be scalars or matrix of the same size as X and Y.
%
%       ARROWS(...,SHAPE) uses relative ratios SHAPE = [HEADW,HEADL,HEADI,LINEW]
%       to adjust head width HEADW, head length HEADL, head inside length HEADI,
%       and segment line width LINEW for an arrow length of 1 or 2*PI for the
%       'Loop' type. Default is SHAPE = [0.2,0.2,0.15,0.05].
%
%       ARROWS(...,'Ref',R) defines a reference length R for which SHAPE
%       parameters applies. Any other lengths will keep the arrow's header size
%       and line width as for the reference.
```

```
%
%       ARROWS(...,'param1',value1,'param2',value2,...) specifies any
%       additionnal properties of the Patch using standard parameter/value
%       pairs, like 'FaceColor','EdgeColor','LineWidth', ...
%
%       H=ARROWS(...) returns graphic's handle of patches.
%
%       Examples:
%
%         arrows(0,0,1,45,'FaceColor','none','LineWidth',3)
%
%         arrows(1,0,1,0,[.2,.4,.2,.02])
%
%         arrows(0,0,-1,45,'Loop')
%
%         [xx,yy] = meshgrid(1:10);
%         arrows(xx,yy,rand(size(xx)),360*rand(size(xx)))
%
%
%       Notes:
%
%       - Arrow shape supposes an equal aspect ratio (axis equal).
%       - To define an arrow without segment line, set HEADI = 1, LINEW = 0,
%         and adjust other shape parameters, e.g., a triangle is defined by
%         SHAPE = [0.5,1,1,0], while a wind arrow is [1,1.5,1,0].
%       - To make arrows in 3-D, use the powerful Matlab's function ROTATE.
%
%       See also PATCH, QUIVER.
%
%       Author: Francois Beauducel
%       Created: 1995-02-03
%       Updated: 2020-06-23

%       Copyright (c) 2020, Francois Beauducel, covered by BSD License.
%       All rights reserved.
%
%       Redistribution and use in source and binary forms, with or without
%       modification, are permitted provided that the following conditions are
%       met:
%
%          * Redistributions of source code must retain the above copyright
%            notice, this list of conditions and the following disclaimer.
%          * Redistributions in binary form must reproduce the above copyright
%            notice, this list of conditions and the following disclaimer in
%            the documentation and/or other materials provided with the
```

```
    distribution
%
%       THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS AS IS
%       AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
%       IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    PURPOSE
%       ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
%       LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
%       CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
%       SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
%       INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
%       CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
%       ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
    THE
%       POSSIBILITY OF SUCH DAMAGE.

if nargin < 4
        error('Not enough input arguments.')
end

if ~isnumeric(x) || ~isnumeric(y) || ~isnumeric(l) || ~isnumeric(az)
        error('X, Y, L/AZ, U/V or R/AZ must be numeric.')
end

n = max([numel(x) numel(y) numel(l) numel(az)]);        % max size of arguments

% default arrow shape
shape = [.2 .2 .15 .05];

% loop arrow length
loopl = 7*pi/4;

% checks the arrow type
types = {'polar','cartesian','loop'};

type = 'polar'; % default arrow type
for i = 1:length(types)
        k = strcmpi(varargin,types{i});
        if any(k)
                type = types{i};
                varargin(k) = [];
        end
end

% reference length option
```

```
k = find(strcmpi(varargin,'ref'));
if ~isempty(k) && nargin>4+k
        refl = varargin{k+1};
        varargin(k+[0,1]) = [];
else
        refl = 0;
end

if ~isempty(varargin) && isnumeric(varargin{1})
        shape = varargin{1}(:)';
        if numel(shape) ~= 4
                error('SHAPE argument must be a 4-scalar vector.')
        end

        % this adjusts head drawing for HEADI = 0 and LINEW > 0
        shape(3) = max(shape(3),shape(4)*shape(2)/shape(1));
        varargin = varargin(2:end);
end

mline = 2;

switch type
        case 'loop'
                mline = 50; % additionnal line points for loop arrow
                az = az*pi/180;
                shape(2:3) = shape(2:3)/loopl;
        case 'cartesian'
                % converts U,V to AZ,L
                [az,l] = cart2pol(l,az);
                az = pi/2 - az;
        otherwise
                az = az*pi/180;
end

% total number of arrow points (see fx vector below)
m = 4 + 2*mline;

% needs to duplicate arguments
x = repmat(x(:)',[m,1]);
y = repmat(y(:)',[m,1]);
l = repmat(l(:)',[m,1]);
az = repmat(az(:)',[m,1]);

% s is a matrix of shape vectors
if refl
```

```
        s = refl*(1./abs(l(1,:))')*shape;
        s(isinf(s)) = 0;        % because 0-length arrows produced Inf shape
            elements...
else
        s = repmat(shape,[n,1]);
end

v0 = zeros(n,1);
v1 = ones(n,1);
vl = linspace(0,1,mline);

% unit length arrow patch
fx = [s(:,1)*[-.5  0  .5]          s(:,4)*.5*[ones(1,mline) 1 -ones(1,mline)]]';
fy = [(1 - s(:,2)) v1 (1 - s(:,2)) (1 - s(:,3))*fliplr(vl) v0 (1 - s(:,3))*vl]';

switch type
        case 'loop'
                th = pi/2 - fy*loopl - az; % arrow length converted to angle loop
                px = l.*cos(th).*(1 + fx);
                py = l.*sin(th).*(1 + fx);
                % head must be redrawn to avoid distortion...
                kh = 1:3; % indexes of the head's 3-points to be moved
                xh = -l(kh,:).*flipud(fx(kh,:));
                yh = -l(kh,:).*((s(:,3) - [s(:,2) v0 s(:,2)])')*loopl;
                a0 = repmat(th(4,:),3,1); % head rotation angle
                % rotates and tranlates the head at the right position
                px(kh,:) = cos(a0).*xh - sin(a0).*yh + repmat(mean(px([4,end],:))
                    ,3,1);
                py(kh,:) = sin(a0).*xh + cos(a0).*yh + repmat(mean(py([4,end],:))
                    ,3,1);

        otherwise
                px = l.*(-fx.*cos(az) + fy.*sin(az));
                py = l.*(fx.*sin(az) + fy.*cos(az));
end

% the beauty of this script: a single patch command to draw all the arrows !
h = patch(px + x,sign(l).*py + y,'k',varargin{:});

if nargout > 0
        varargout{1} = h;
end

function h = otbot_circle_v2(x,y,r,fillcolor)
th = 0:pi/50:2*pi;
```

```
xunit = r * cos(th) + x;
yunit = r * sin(th) + y;
h = plot(xunit, yunit,'LineWidth', 0.75, 'Color', [0 0 0]);

srtc = strcmp(fillcolor,'NO');
if srtc == 0
    h = fill(xunit,yunit,fillcolor);
end
end

function [] = multi_circles(XYZmatrix,radius,fillcolor)
%MULTI_CIRCLES Function to plot multiple circles at desired positions
%   This function is designed to plot a circle in each specified position
%   with the same radius and the same color. Warning: Z input will be
%   dismissed for now

Dim = size(XYZmatrix);


hold on
for iaux1 = 1:Dim(2)
    otbot_circle_v2(XYZmatrix(1,iaux1),XYZmatrix(2,iaux1),radius,fillcolor);
end
hold off

end
```

A.3 Parameter indentification routines

This section shows the code used in dynamic simulation. The scheme of all the programs used is described in Fig. A.2. We start by running all the dynamic simulation routines in order to perform what can be named *reality* experiment, from which we collect all the required data for the identification. It is in this step where we include the noise in the data. Once we have completed the experiment and the sampling we proceed to execute the grey-box identification routines by executing the first script named `a1_main.m` where the general settings for the estimation are prepared, this includes defining which parameters will be identified and which will not, as well as the directories and folders where the results will be saved. This script this script automatically executes the next part of the code, another script called `submain_nlgreyest.m`, which calls the two main pieces of code named `a_setup_otbot_gbme.m`, `b_param_est_otbot.m`.

The first one is in charge of to load the data of the experiment and create the `iddata` object that `nlgreyest` will require to work with. This data object contains important info such as the sample period used, indicates which elements of the array are inputs to the system and which are outputs, sets up the units, sets what type of interpolation has been used for the actions input (i.e. zero-order hold), asks the user if needs to display and save the data plots, sets up the initial guesses for each of the parameters that will be estimated, defines the `idnlgrey` object which

will contain all the configurations of the model, sets the initial states of the system, defines the system as a continuous time and defines the ranges that limit the values that the estimations can take.

The second one is more user centered, and is in charge of asking the user if needs to visualise any particular configuration of the objects created to check if everything is configured as it should be before running the parameter estimation. Also fixes some of the options for the estimation program such as the display of the status of the estimation, the search methods used, the algorithm and the tolerances. At the final part of this file if everything the final part of this file if everything has been validated is where the parameter estimation is executed.

Subsequently, it returns to the `submain_nlgreyest.m` script, where it asks the user how the results are going to be displayed and saved. Finally we only have to make if desired a sensitivity analysis of the results by accumulating several experiments, this can be done thanks to the fact that the previous code saves the data in `.mat` files that can be loaded later for analysis.

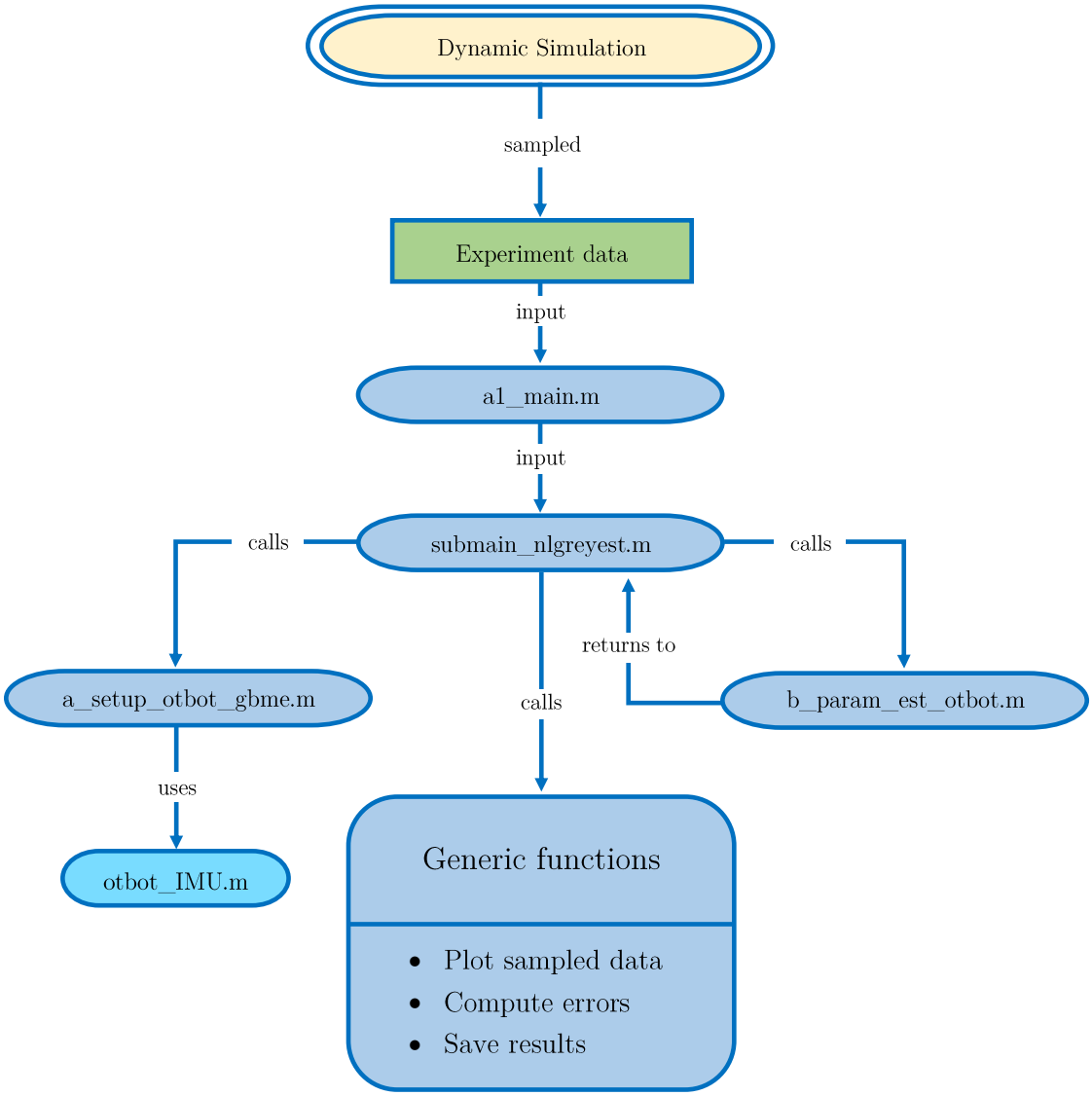


Figure A.2: Organization chart that describes the structure of the code for parameter indentification routines.

A.3.1 Main parameter identification (a1_main.m)

```
%% This is the main script to call the oders and perform de model estimation

% Preset which parameters are fixed and which will be estimated
FixorNot(1,1) = false;   % I_b [kg*m^2] % Paramater we want to estimate
FixorNot(2,1) = true;    % I_p [kg*m^2]
FixorNot(3,1) = true;    % I_a [kg*m^2]
FixorNot(4,1) = true;    % I_t [kg*m^2]
FixorNot(5,1) = true;    % l_1 [m]
FixorNot(6,1) = true;    % l_2 [m]
FixorNot(7,1) = false;   % m_b [kg] % Paramater we want to estimate
FixorNot(8,1) = true;    % m_w [kg]
FixorNot(9,1) = true;    % m_p [kg]
FixorNot(10,1) = false;  % x_G [m]  % Paramater we want to estimate
FixorNot(11,1) = false;  % y_G [m]  % Paramater we want to estimate
FixorNot(12,1) = true;   % x_F [m]
FixorNot(13,1) = true;   % y_F [m]
FixorNot(14,1) = true;   % r   [m]
FixorNot(15,1) = true;   % b_r [kg*m^2*s^-1]
FixorNot(16,1) = true;   % b_l [kg*m^2*s^-1]
FixorNot(17,1) = true;   % b_p [kg*m^2*s^-1]

%% Set the directories for saving the results

DirectoryName1 = 'Default(CoM0-0.5l1)/';
% DirectoryName2 = '1.02 - 3xCte_T_FND_3Snoise [0.01]/';
DirectoryName2 = 'TEST/';

%% Set the name of the sampling plots

CN1_SP = '  Cte_T(6,-10,6)_1.02_3s_3SNoise.png'; % Name the image

DirName_SP = '3s'; % Save the sample plots inside the desired folder

FullDirNameSP = strcat('Exp/',DirectoryName1,DirectoryName2, ...
    'Sample_Plots/',DirName_SP);

[MKdirStatusSP, msgSP] = mkdir(FullDirNameSP);

%% Set algorithm search options
NLGreySearchConfig = 0; % With this option the user decides the set of
                        % options concerning the Search Method and the algorithm
                        % (NLGreySearchConfig == 0)> Default setttings
                        % SearchMethod is lsqnonlin and the
```

```
                        % Algorithm is rust-region-reflective

                        % (NLGreySearchConfig == 1)> Settings will be set
                        % to work with fmincon usinf the interior-point
                        % algorithm
```

```
%% Set up the saving directory for the excel Tables of results

ExcelFileName = 'mcIcxByB';

DirectoryName3 = 'mc_Ic_xB_yB 3s';

FullDirNameExcel = strcat('Exp/',DirectoryName1,DirectoryName2, ...
    'Results/',DirectoryName3,'/',ExcelFileName);
PartialDirNameExcel = strcat('Exp/',DirectoryName1,DirectoryName2, ...
    'Results/',DirectoryName3);
[MKdirStatusExcel,msgExcel] = mkdir(PartialDirNameExcel);
```

```
%% Runing the submain script
submain_nlgreyest
```

A.3.2 Main nlgreyest (submain_nlgreyest.m)

```
%% Step 1 Load data and set-up the idngrey
% we call the script to do so
a_setup_otbot_gbme
```

```
%% Step 2 - Set up and Model Identification
% In this section we set up all the free parameters the fixed ones and same
% for inicial states. Also we display all the configurations for the user
% in order to check if everything has been set correctly.
b_param_est_otbot
```

```
%% Displaying the results
if CESFlag == 1
    [pvec, pvec_sd] = getpvec(nlgrm);
    load('real_pars_data.mat')
    ParameterRealValues = real_pars;
    ParameterInitialGuesses = pvecinit;
    ParameterFinalValues = pvec;

    prompt = 'Estimation terminated successfully! Do you want to display' ...
        ' the results? Y/N [Y]: ';
    mainstrinput = input(prompt,'s');
    if isempty(mainstrinput)
        mainstrinput = 'Y';
```

```
end

ParameterName = {'I_b'; 'I_p'; 'I_a'; 'I_t'; 'l_1'; 'l_2'; 'm_b'; 'm_w'; ...
    'm_p'; 'x_G'; 'y_G'; 'x_F'; 'y_F'; 'r'; 'b_r'; 'b_l'; 'b_p'};
RelativeError_Percentage = rel_error(ParameterRealValues,
   ParameterFinalValues);
StDev = pvec_sd;
Units = {'kg*m^2'; 'kg*m^2'; 'kg*m^2'; 'kg*m^2'; 'm'; 'm'; 'kg'; 'kg'; ...
    'kg'; 'm'; 'm'; 'm'; 'm'; 'm'; 'kg*m^2*s^-1'; 'kg*m^2*s^-1'; ...
    'kg*m^2*s^-1'};
ResultsTable = table(ParameterName, Units, ParameterRealValues, ...
    ParameterInitialGuesses, ParameterFinalValues,  ...
    RelativeError_Percentage, StDev );

if strcmp(mainstrinput, 'Y') || strcmp(mainstrinput, 'y')
    disp('Displaying a Table with the final values of the parameters')
    disp(ResultsTable)
elseif strcmp(mainstrinput, 'N') || strcmp(mainstrinput, 'n')
    disp('The display of results will be omitted')
else
    disp('Unrecognised input, omitting the display of results')
end

% Save everything?
prompt = 'Do you want to save everything? Y/N [Y]: ';
mainstrinput = input(prompt,'s');
if isempty(mainstrinput)
    mainstrinput = 'Y';
end

if strcmp(mainstrinput, 'Y') || strcmp(mainstrinput, 'y')
    savefinalparams
    save_exceltable
    save_sensitivitydata
    save_Report
elseif strcmp(mainstrinput, 'N') || strcmp(mainstrinput, 'n')
    % Save the final values of the parameters
    prompt = 'Do you want to save final values of the Parameters in' ...
        ' .mat file? Y/N [N]: ';
    mainstrinput = input(prompt,'s');
    if isempty(mainstrinput)
        mainstrinput = 'N';
    end

    if strcmp(mainstrinput, 'Y') || strcmp(mainstrinput, 'y')
```

```
    savefinalparams
elseif strcmp(mainstrinput, 'N') || strcmp(mainstrinput, 'n')
    disp('The table of results will not be saved')
else
    disp('Unrecognised input, omitting the display of results')
end

% Save the table in Excel

prompt = 'Do you want to save the results table in Excel? Y/N [N]: ';
mainstrinput = input(prompt,'s');
if isempty(mainstrinput)
    mainstrinput = 'N';
end

if strcmp(mainstrinput, 'Y') || strcmp(mainstrinput, 'y')
    save_exceltable
elseif strcmp(mainstrinput, 'N') || strcmp(mainstrinput, 'n')
    disp('The table of results will not be saved')
else
    disp('Unrecognised input, omitting the display of results')
end

% Save the data for sensitivity analisis

prompt = 'Do you want to save the data for the sensitivity' ...
    ' analisis? Y/N [N]: ';
mainstrinput = input(prompt,'s');
if isempty(mainstrinput)
    mainstrinput = 'N';
end

if strcmp(mainstrinput, 'Y') || strcmp(mainstrinput, 'y')
    save_sensitivitydata
elseif strcmp(mainstrinput, 'N') || strcmp(mainstrinput, 'n')
    disp('The data for sensitivity analisis will not be saved')
else
    disp('Unrecognised input, data for sensitivity analisis will' ...
        ' not be saved')
end

% Save Results Report

prompt = 'Do you want to save results report? Y/N [N]: ';
mainstrinput = input(prompt,'s');
```

```
        if isempty(mainstrinput)
            mainstrinput = 'N';
        end

        if strcmp(mainstrinput, 'Y') || strcmp(mainstrinput, 'y')
            save_Report
        elseif strcmp(mainstrinput, 'N') || strcmp(mainstrinput, 'n')
            disp('The results report will not be saved')
        else
            disp('Unrecognised input, results report will not be saved')
        end
    else
       disp('Unrecognised input, nothing will be saved')
    end
end
```

A.3.3 Set greybox objects (a_setup_otbot_gbme.m)

```
%% 1. Load measured data.

% The measured angular displacement data is loaded and saved as data,
% an iddata object with a sample time of 0.1 seconds. The set command is
% used to specify data attributes such as the output name,  output unit,
% and the start time and units of the time vector.
prompt = 'Please introduce the sample time of the data [0.01s]: ';
TSinput = input(prompt);
if isempty(TSinput)
    TSinput = 0.01;
end

% Set the deviation of the initial guess with respect the real values
prompt = 'Please introduce the deviation of the parameters for' ...
    'the initial guess [50%]: ';
TSinputDev = input(prompt);
if isempty(TSinputDev)
    TSinputDev = 50;
end

IGD = 1-TSinputDev/100;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
load('Otbot_Data.mat'); % Notice that this is not otbot data and needs to
                        % be changed
data = iddata(y_out_data, u_out_data, TSinput, ...
    'Name', 'Otbot_Data'); % Output signal of the system y
```

```
                                        % Input signal of the system u
                                        % Sample time Ts

% Specify input and output names, start time and time units
data.InputName = {'Right wheel torque';      ... % u(1).
                  'Left wheel torque';       ... % u(2).
                  'Platform torque'};            % u(3).
data.InputUnit =  {'Nm';'Nm';'Nm'};
data.OutputName = {'alpha_dot';'x_ddot';'y_ddot'};
data.OutputUnit = {'rad/s';'m/s^2';'m/s^2'};
data.Tstart = 0;
data.TimeUnit = 's';

% Specify intersample behaviour for transformations between discrete time
% and continuous time.
data.InterSample = {'zoh';  % Behaviour for the first input u(1) —
                            % Right wheel torque
                    'zoh';  % Behaviour for the second input u(2) —
                            % Left wheel torque
                    'zoh'}; % Behaviour for the third input u(3) —
                            % Platform torque

%———Plot the Data———%
prompt = 'Do you want to plot the data that has been loaded? Y/N [N]: ';
stringput = input(prompt,'s');
if isempty(stringput)
    stringput = 'N';
end

if strcmp(stringput, 'Y') || strcmp(stringput, 'y')
    plottingsampledata
    prompt = 'Do you want to save the plots? Y/N [N]: '; % Saveing the plots
    stringput = input(prompt,'s');
    if isempty(stringput)
        stringput = 'N';
    end

    if strcmp(stringput, 'Y') || strcmp(stringput, 'y')
        save_sampleplots
    elseif strcmp(stringput, 'N') || strcmp(stringput, 'n')
        disp('The plots will not be saved')
    else
        disp('Unrecognised input, the plots will not be saved')
    end
elseif strcmp(stringput, 'N') || strcmp(stringput, 'n')
```

```
    disp('The plots of the loaded data will be omitted')
else
    disp('Unrecognised input, the plots of the loaded data will be omitted')
end
%-------------------------%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% 2. Represent the model using an idnlgrey object
load('real_pars_data.mat')

% Create a nonlinear grey-box model associated with the otbot_m function.

% Central moment of inertia of the full chassis (chassis body plus wheels)
% about axis 3 [kg*m^2]
I_b = real_pars(1) *(IGD*double(~FixorNot(1,1)) + double(FixorNot(1,1)));
% Central moment of inertia of the platform body about axis 3'' [kg*m^2]
I_p = real_pars(2) *(IGD*double(~FixorNot(2,1)) + double(FixorNot(2,1)));
% Axial moment of inertia of one wheel [kg*m^2]
I_a = real_pars(3) *(IGD*double(~FixorNot(3,1)) + double(FixorNot(3,1)));
% Twisting moment of inertia of one wheel [kg*m^2] (Will be set to 0 when
% identifying to add the value to I_b)
I_t = real_pars(4) *(IGD*double(~FixorNot(4,1)) + double(FixorNot(4,1)));


% Pivot offset relative to the wheels axis [m]
l_1 = real_pars(5) *(IGD*double(~FixorNot(5,1)) + double(FixorNot(5,1)));
% One half of the wheels separation [m]
l_2 = real_pars(6) *(IGD*double(~FixorNot(6,1)) + double(FixorNot(6,1)));


% Mass of the full chassis (chassis body plus wheels) [kg]
m_b = real_pars(7) *(IGD*double(~FixorNot(7,1)) + double(FixorNot(7,1)));
% Mass of one wheel [kg] (Will be set to 0 when identifying to add the value to
    m_b)
m_w = real_pars(8) *(IGD*double(~FixorNot(8,1)) + double(FixorNot(8,1)));
% Mass of the platform [kg]
m_p = real_pars(9) *(IGD*double(~FixorNot(9,1)) + double(FixorNot(9,1)));



%------- This coordinates are 0 so that they will be treated in special way
% x coord of the c.o.m. of the full chassis (chassis body plus wheels)
% in the chassis frame [m]
```

```
x_G = real_pars(10)*(IGD*double(~FixorNot(10,1)) + double(FixorNot(10,1)));
% y coord of the c.o.m. of the full chassis (chassis body plus wheels)
% in the chassis frame [m]
y_G = 0.5*((1-IGD)*double(~FixorNot(11,1)) + 0*double(FixorNot(11,1)));


% x coord of the c.o.m. of the platform body in the platform frame [m]
x_F = 0.5*((1-IGD)*double(~FixorNot(12,1)) + 0*double(FixorNot(12,1)));
% y coord of the c.o.m. of the platform body in the platform frame [m]
y_F = 0.5*((1-IGD)*double(~FixorNot(13,1)) + 0*double(FixorNot(13,1)));
%--------------------

% Wheel radius [m]
r = real_pars(14) *(IGD*double(~FixorNot(14,1)) + double(FixorNot(14,1)));

% Friction coefficient of right wheel [kg*m^2*s^-1]
b_r = real_pars(15) * (IGD*double(~FixorNot(15,1)) + double(FixorNot(15,1)));

% Friction coefficient of left wheel [kg*m^2*s^-1]
b_l = real_pars(16) * (IGD*double(~FixorNot(16,1)) + double(FixorNot(16,1)));

% Frictin coefficient of pivot joint [kg*m^2*s^-1]
b_p = real_pars(17) * (IGD*double(~FixorNot(17,1)) + double(FixorNot(17,1)));


FileName       = 'otbot_IMU';        % File describing the model structure.
order          =  [3 3 12];          % Order: Number of outputs, inputs,
                                     % and states of the model [Ny,Nu,Nx]
parameters     = {I_b,I_p,I_a,I_t,l_1,l_2,m_b,m_w,m_p,x_G,y_G,x_F,y_F,r, ...
                  b_r,b_l,b_p};  % Parameters of the model


%--------Initial States--------%
prompt = 'HOW DO YOU WANT TO SET THE INITAL STATE? 0/E [E]: ';
mainstrinput = input(prompt,'s');
if isempty(mainstrinput)
    mainstrinput = 'E';
end

if strcmp(mainstrinput, '0') || strcmp(mainstrinput, 'zero')
    initial_states = zeros(12,1);
elseif strcmp(mainstrinput, 'E') || strcmp(mainstrinput, 'e')
    disp('The initial state will be set to the values of initial_states.mat')
    load('initial_states.mat')
    initial_states = xs0;
```

```
else
    disp('Unrecognised input, initial states will be set to 0')
    initial_states = zeros(12,1);
end
%--------------------------------%

Ts              = 0;                % Specify as continuous system.
nlgrm = idnlgrey(FileName,order,parameters,initial_states,Ts,'Name', ...
                 'Otbot_IMU_GM');

% Specify input and output names, and units.
nlgrm.InputName  = {'Right wheel torque';    ... % u(1).
                    'Left wheel torque';     ... % u(2).
                    'Platform torque'};          % u(3).

nlgrm.InputUnit  = {'Nm';'Nm';'Nm'};

nlgrm.OutputName = {'alpha_dot';'x_ddot';'y_ddot'};

nlgrm.OutputUnit = {'rad/s';'m/s2';'m/s2'};

set(nlgrm,'TimeUnit', 's');

% Specify names and units of the initial states and parameters
% Initial States names and units
nlgrm = setinit(nlgrm, 'Name', {'x';'y';'alpha';'varphi_r';'varphi_l'; ...
                                'varphi_p'; 'x_dot';'y_dot';'alpha_dot'; ...
                                'varphi_dot_r';'varphi_dot_l';'varphi_dot_p'});

nlgrm = setinit(nlgrm, 'Unit', {'m';'m';'rad';'rad';'rad';'rad'; ...
                                'm/s';'m/s';'rad/s';'rad/s';'rad/s';'rad/s'});

% Inital parameters names and units
nlgrm = setpar(nlgrm, 'Name', {'Inertia of the chassis body';               ... %
    I_b
                               'Inertia of the platform body';              ... %
                                   I_p
                               'Axial moment of inertia of one wheel';      ... %
                                   I_a
                               'Twisting moment of inertia of one wheel';   ... %
                                   I_t
                               'Pivot offset';                              ... %
                                   l_1
                               'One half of the wheels separation';         ... %
                                   l_2
```

```
                                'Mass of the chassis base';                  ... %
                                    m_b
                                'Mass of one wheel';                         ... %
                                    m_w
                                'Mass of the platform';                      ... %
                                    m_p
                                'x coord of the c.o.m. of the chassis body'; ... %
                                    x_G
                                'y coord of the c.o.m. of the chassis body'; ... %
                                    y_G
                                'x coord of the c.o.m. of the plat. body';   ... %
                                    x_F
                                'y coord of the c.o.m. of the plat. body';   ... %
                                    y_F
                                'Wheel radius';                              ... %
                                    r
                                'Viscous friction right wheel';              ... %
                                    b_r
                                'Viscous friction left wheel';               ... %
                                    b_l
                                'Viscous friction pivot joint'});                %
                                    b_p

nlgrm = setpar(nlgrm, 'Unit', {'kg*m2';'kg*m2';'kg*m2';'kg*m2'; ...
                               'm';'m';'kg';'kg';'kg';'m';'m';'m';'m'; ...
                               'm';'kg*m2*s-1';'kg*m2*s-1';'kg*m2*s-1'});

nlgrm = setpar(nlgrm, 'Minimum', num2cell(  [eps(0)*ones(3,1);  ... % Moments of
    Inertia are > 0!
                                            -1;                 ... % Moment of
                                                Inertia of I_t will be set to
                                                zero to make I_c = I_b
                                            eps(0)*ones(2,1);   ... % Lenghts are
                                                 > 0
                                            eps(0)*ones(1,1);   ... % Masses are
                                                 > 0
                                            -1;                 ... % Mass of the
                                                 wheel m_w will be set to 0 to
                                                make m_c = m_b
                                            eps(0)*ones(1,1);   ... % Masses are
                                                 > 0
                                            -100*ones(4,1);     ... % We set a
                                                min to the coordinates to use
                                                fmincon
                                            eps(0);             ... % The value
```

```
                                                    of the wheel radius is > 0!
                                                -100;               ... % Minimum
                                                    value for b_r not very rellevant
                                                -100;               ... % Minimum
                                                    value for b_l not very rellevant
                                                -100]));                % Minimum
                                                    value for b_p not very rellevant

nlgrm = setpar(nlgrm, 'Maximum', num2cell(  [100*ones(4,1);  ... % Moments of
    Inertia arbitrari maximum to use fmincon
                                                100*ones(2,1);  ... % Lenghts
                                                    arbitrari maximum to use fmincon
                                                1e6*ones(3,1);  ... % Masses
                                                    arbitrari maximum to use fmincon
                                                100*ones(4,1);  ... % We set a
                                                    maximum to the coordinates to
                                                    use fmincon
                                                30;             ... % Set a maximum
                                                    value for the wheel radius to
                                                    use fmincon
                                                100;            ... % We now set a
                                                    maximum for b_r even that it won
                                                    't be very important
                                                100;            ... % We now set a
                                                    maximum for b_l even that it won
                                                    't be very important
                                                100]));             % We now set a
                                                    maximum for b_p even that it won
                                                    't be very important
```

A.3.4 Extra configurations and identification (b_param_est_otbot.m)

```
%% Specify Initial Sate estimation
% By default the inital states will not be estimated if we want to estimate
% them we must set it up.

nlgrm.InitialStates(1).Fixed = true;  % x          [m]
nlgrm.InitialStates(2).Fixed = true;  % y          [m]
nlgrm.InitialStates(3).Fixed = true;  % alpha      [rad]
nlgrm.InitialStates(4).Fixed = true;  % varphi_r   [rad]
nlgrm.InitialStates(5).Fixed = true;  % varphi_l   [rad]
nlgrm.InitialStates(6).Fixed = true;  % varphi_p   [rad]
nlgrm.InitialStates(7).Fixed = true;  % x_dot      [m/s]
nlgrm.InitialStates(8).Fixed = true;  % y_dot      [m/s]
nlgrm.InitialStates(9).Fixed = true;  % alpha_dot  [rad/s]
```

```
nlgrm.InitialStates(10).Fixed = true; % varphi_r   [rad/s]
nlgrm.InitialStates(11).Fixed = true; % varphi_l   [rad/s]
nlgrm.InitialStates(12).Fixed = true; % varphi_p   [rad/s]

%% Specify known parameters
% Specify which parameters are alredy known and which wants we need to
% estimate

nlgrm.parameters(1).Fixed = FixorNot(1,1);  % I_b [kg*m^2]
nlgrm.parameters(2).Fixed = FixorNot(2,1);  % I_p [kg*m^2]
nlgrm.parameters(3).Fixed = FixorNot(3,1);  % I_a [kg*m^2]
nlgrm.parameters(4).Fixed = FixorNot(4,1);  % I_t [kg*m^2]
nlgrm.parameters(5).Fixed = FixorNot(5,1);  % l_1 [m]
nlgrm.parameters(6).Fixed = FixorNot(6,1);  % l_2 [m]
nlgrm.parameters(7).Fixed = FixorNot(7,1);  % m_b [kg]
nlgrm.parameters(8).Fixed = FixorNot(8,1);  % m_w [kg]
nlgrm.parameters(9).Fixed = FixorNot(9,1);  % m_p [kg]
nlgrm.parameters(10).Fixed = FixorNot(10,1); % x_G [m]
nlgrm.parameters(11).Fixed = FixorNot(11,1); % y_G [m]
nlgrm.parameters(12).Fixed = FixorNot(12,1); % x_F [m]
nlgrm.parameters(13).Fixed = FixorNot(13,1); % y_F [m]
nlgrm.parameters(14).Fixed = FixorNot(14,1); % r   [m]
nlgrm.parameters(15).Fixed = FixorNot(15,1); % b_r [kg*m^2*s^-1]
nlgrm.parameters(16).Fixed = FixorNot(16,1); % b_l [kg*m^2*s^-1]
nlgrm.parameters(17).Fixed = FixorNot(17,1); % b_p [kg*m^2*s^-1]

%% Visualise model configuration

prompt = 'Do you want visualise extra configurations of the model? Y/N [N]: ';
strvis = input(prompt,'s');
if isempty(strvis)
    strvis = 'N';
end

if strcmp(strvis, 'Y') || strcmp(strvis, 'y')

    % View the inital model
    % a. Get basic information about the model
    % The Otbot has 12 states and 15 model parameters
    disp('This is the dimention of our Otbot model');
    size(nlgrm)

    % b. View the initial states and parameters
    % First we view the initial States
    prompt = 'Do you want to display the inital states configuration? Y/N [N]: ';
```

```
stringput = input(prompt,'s');
if isempty(stringput)
    stringput = 'N';
end

if strcmp(stringput, 'Y') || strcmp(stringput, 'y')
    disp('Intitial state of x')
    nlgrm.InitialStates(1)

    disp('Intitial state of y')
    nlgrm.InitialStates(2)

    disp('Intitial state of alpha')
    nlgrm.InitialStates(3)

    disp('Intitial state of varphi_r')
    nlgrm.InitialStates(4)

    disp('Intitial state of varphi_l')
    nlgrm.InitialStates(5)

    disp('Intitial state of varphi_p')
    nlgrm.InitialStates(6)

    disp('Intitial state of x_dot')
    nlgrm.InitialStates(7)

    disp('Intitial state of y_dot')
    nlgrm.InitialStates(8)

    disp('Intitial state of alpha_dot')
    nlgrm.InitialStates(9)

    disp('Intitial state of varphi_dot_r')
    nlgrm.InitialStates(10)

    disp('Intitial state of varphi_dot_l')
    nlgrm.InitialStates(11)

    disp('Intitial state of varphi_dot_p')
    nlgrm.InitialStates(12)
elseif strcmp(stringput, 'N') || strcmp(stringput, 'n')
    disp('Omitting the display of the inital states')
else
    disp('Input not recognised, omitting display of the initial states')
```

```
end

% Then the parameters
prompt = 'Do you want to display the parameters configuration? Y/N [N]: ';
strinput = input(prompt,'s');
if isempty(strinput)
    strinput = 'N';
end

if strcmp(strinput, 'Y') || strcmp(strinput, 'y')
    nlgrm.Parameters(1)
    nlgrm.Parameters(2)
    nlgrm.Parameters(3)
    nlgrm.Parameters(4)
    nlgrm.Parameters(5)
    nlgrm.Parameters(6)
    nlgrm.Parameters(7)
    nlgrm.Parameters(8)
    nlgrm.Parameters(9)
    nlgrm.Parameters(10)
    nlgrm.Parameters(11)
    nlgrm.Parameters(12)
    nlgrm.Parameters(13)
    nlgrm.Parameters(14)
    nlgrm.Parameters(15)
    nlgrm.Parameters(16)
    nlgrm.Parameters(17)
elseif strcmp(strinput, 'N') || strcmp(strinput, 'n')
    disp('Omitting display of parameters')
else
    disp('Input not recognised, omitting display of parameters')
end

% Then the initial states that are fixed
prompt = 'Do you want to display which initial states are Fixed? Y/N [N]: ';
strinput = input(prompt,'s');
if isempty(strinput)
    strinput = 'N';
end

if strcmp(strinput, 'Y') || strcmp(strinput, 'y')
    getinit(nlgrm,'Fixed')
elseif strcmp(strinput, 'N') || strcmp(strinput, 'n')
    disp('Omitting display of fixed initial states')
else
```

```
        disp('Input not recognised, omitting display of fixed initial states')
    end


    % Then the minimum values for the parameters
    prompt = 'Do you want to display the minimun allowed values for the' ...
             'parameters? Y/N [N]: ';
    strinput = input(prompt,'s');
    if isempty(strinput)
        strinput = 'N';
    end

    if strcmp(strinput, 'Y') || strcmp(strinput, 'y')
        getpar(nlgrm,'Min')
    elseif strcmp(strinput, 'N') || strcmp(strinput, 'n')
        disp('Omitting display of minimun allowed values for the parameters')
    else
        disp('Input not recognised, omitting display of minimun allowed' ...
             'values for the parameters')
    end

    % Obtain basic information about the object
    disp('Displaying basic information about the idngrey object')
    disp(nlgrm)

    prompt = 'Do you want extra information about the current object? Y/N [N]: ';
    strinput = input(prompt,'s');
    if isempty(strinput)
        strinput = 'N';
    end

    if strcmp(strinput, 'Y') || strcmp(strinput, 'y')
        disp('Running the commands get and present to gain extra information')
        get(nlgrm)
        present(nlgrm)
    elseif strcmp(strinput, 'N') || strcmp(strinput, 'n')
        disp('Omitting display of extra information of the idngrey object')
    else
        disp('Input not recognised, omitting display of extra information' ...
             'of the idngrey object')
    end
elseif strcmp(strvis, 'N') || strcmp(strvis, 'n')
    disp('Omitting the display of any configuration')
else
    disp('Input not recognised, omitting display of any configuration')
```

```
end

%% Set up the nlgreyest options
% In this case we only want to choose to display the estimation progress.
% But there are a lot more options that can be changed as desired.
% See nlgreyestOptions.
opt = nlgreyestOptions('Display','on');

switch NLGreySearchConfig
    case 0
        disp('Runing nlgreyest with default searching configuration (lsqnonlin)')
    case 1
        disp('Runing nlgreyest with fmincon & interior-point')
        opt.SearchMethod = 'fmincon';
        opt.SearchOptions.Algorithm = 'interior-point';
    otherwise
        disp('WARNING: this NLGreySearchConfig does not exist, runing' ...
             'with the default configuration (lsqnonlin)')
end
% opt.SearchOptions.FunctionTolerance = 1e-30;

%% Estimate the model
% The nlgreyest command updates the parameter of nlgrm.
prompt = 'Do you want to continue with the execution of the script? Y/N [Y]: ';
stringput = input(prompt,'s');
if isempty(stringput)
    stringput = 'Y';
end

if strcmp(stringput, 'Y') || strcmp(stringput, 'y')
    [pvecinit, pvec_sdinit] = getpvec(nlgrm);
    tStart = tic;
    nlgrm = nlgreyest(data,nlgrm,opt);
    tEnd = toc(tStart);
    CESFlag = 1;
elseif strcmp(stringput, 'N') || strcmp(stringput, 'n')
    disp('The execution will be terminated')
    CESFlag = 0;
else
    disp('Unrecognised input, terminating the execution')
    CESFlag = 0;
end
```

A.3.5 Dynamic model for identification

```
function [xdot,yout] = otbot_IMU(t, xs, u, I_b, I_p, I_a, I_t, l_1, ...
    l_2, m_b, m_w, m_p, x_G, y_G, x_F, y_F, r, b_r, b_l, b_p, varargin)
%OTBOT_M Summary of this function goes here
%  Function containing the equations of motion of Otbot

%--------------- List of state variables ---------------%
% Note how some of this state variables (the ones commented out) are not
% used to compute the equations of motion of the system.

% x = xs(1);
% y = xs(2);
alpha = xs(3);
% varphi_r = xs(4);
% varphi_l = xs(5);
varphi_p = xs(6);

% x_dot = xs(7);
% y_dot = xs(8);
alpha_dot = xs(9);
varphi_dot_r = xs(10);
varphi_dot_l = xs(11);
varphi_dot_p = xs(12);

%--------------- Create viscous friction term ---------------%

tau_friction(1:3,1) = zeros(3,1);
        tau_friction(4,1) = -b_r*varphi_dot_r; % tau_friction_right
        tau_friction(5,1) = -b_l*varphi_dot_l; % tau_friction_left
        tau_friction(6,1) = -b_p*varphi_dot_p; % tau_friction_pivot

%--------------- Create system matrices ---------------%

MIIKs = reshape([(l_1.*cos(alpha-varphi_p)-l_2.*sin(alpha-varphi_p)) ...
    ./(l_1.*r),(l_1.*cos(alpha-varphi_p)+l_2.*sin(alpha-varphi_p))./ ...
    (l_1.*r),sin(alpha-varphi_p)./l_1,(l_2.*cos(alpha-varphi_p)+l_1.* ...
    sin(alpha-varphi_p))./(l_1.*r),-(l_2.*cos(alpha-varphi_p)-l_1.* ...
    sin(alpha-varphi_p))./(l_1.*r),-cos(alpha-varphi_p)./l_1,0.0,0.0,1.0],[3,3]);

M_bars2 = reshape([((I_a.*l_2.^2.*sin(alpha-varphi_p).*2.0+I_b.*r.^2.* ...
    sin(alpha-varphi_p)+I_t.*r.^2.*sin(alpha-varphi_p).*2.0+l_1.^2.*m_b.* ...
    r.^2.*sin(alpha-varphi_p)+l_1.^2.*m_p.*r.^2.*sin(alpha-varphi_p)+ ...
    l_2.^2.*m_w.*r.^2.*sin(alpha-varphi_p).*2.0-I_a.*l_1.*l_2.* ...
    cos(alpha-varphi_p).*2.0+m_b.*r.^2.*x_G.^2.*sin(alpha-varphi_p)+m_b.* ...
    r.^2.*y_G.^2.*sin(alpha-varphi_p)+l_1.*m_p.*r.^2.*y_F.*cos(alpha)+ ...
    l_1.*m_p.*r.^2.*x_F.*sin(alpha)-l_1.*l_2.*m_b.*r.^2.*cos(alpha-varphi_p) ...
```

```
    -l_1.*l_2.*m_p.*r.^2.*cos(alpha-varphi_p)-l_1.*l_2.*m_w.*r.^2.* ...
    cos(alpha-varphi_p).*2.0+l_1.*m_b.*r.^2.*y_G.*cos(alpha-varphi_p)+ ...
    l_1.*m_b.*r.^2.*x_G.*sin(alpha-varphi_p).*2.0-l_2.*m_b.*r.^2.*y_G.* ...
    sin(alpha-varphi_p)).*(-1.0./2.0)./(l_1.*l_2.*r),(I_a.*l_2.^2.* ...
    sin(alpha-varphi_p).*2.0+I_b.*r.^2.*sin(alpha-varphi_p)+I_t.*r.^2.* ...
    sin(alpha-varphi_p).*2.0+l_1.^2.*m_b.*r.^2.*sin(alpha-varphi_p)+l_1.^2.* ...
    m_p.*r.^2.*sin(alpha-varphi_p)+l_2.^2.*m_w.*r.^2.*sin(alpha-varphi_p).* ...
    2.0+I_a.*l_1.*l_2.*cos(alpha-varphi_p).*2.0+m_b.*r.^2.*x_G.^2.* ...
    sin(alpha-varphi_p)+m_b.*r.^2.*y_G.^2.*sin(alpha-varphi_p)+l_1.*m_p.* ...
    r.^2.*y_F.*cos(alpha)+l_1.*m_p.*r.^2.*x_F.*sin(alpha)+l_1.*l_2.*m_b.* ...
    r.^2.*cos(alpha-varphi_p)+l_1.*l_2.*m_p.*r.^2.*cos(alpha-varphi_p)+l_1.* ...
    l_2.*m_w.*r.^2.*cos(alpha-varphi_p).*2.0+l_1.*m_b.*r.^2.*y_G.* ...
    cos(alpha-varphi_p)+l_1.*m_b.*r.^2.*x_G.*sin(alpha-varphi_p).* ...
    2.0+l_2.*m_b.*r.^2.*y_G.*sin(alpha-varphi_p)),(l_1.*l_2.*r.*2.0),-m_p.* ...
    (y_F.*cos(alpha)+x_F.*sin(alpha)),(I_a.*l_2.^2.*cos(alpha-varphi_p).* ...
    2.0+I_b.*r.^2.*cos(alpha-varphi_p)+I_t.*r.^2.*cos(alpha-varphi_p).* ...
    2.0+l_1.^2.*m_b.*r.^2.*cos(alpha-varphi_p)+l_1.^2.*m_p.*r.^2.* ...
    cos(alpha-varphi_p)+l_2.^2.*m_w.*r.^2.*cos(alpha-varphi_p).*2.0+m_b.* ...
    r.^2.*x_G.^2.*cos(alpha-varphi_p)+m_b.*r.^2.*y_G.^2.*cos(alpha-varphi_p)+ ...
    I_a.*l_1.*l_2.*sin(alpha-varphi_p).*2.0+l_1.*m_p.*r.^2.*x_F.* ...
    cos(alpha)-l_1.*m_p.*r.^2.*y_F.*sin(alpha)+l_1.*m_b.*r.^2.*x_G.* ...
    cos(alpha-varphi_p).*2.0-l_2.*m_b.*r.^2.*y_G.*cos(alpha-varphi_p)+l_1.* ...
    l_2.*m_b.*r.^2.*sin(alpha-varphi_p)+l_1.*l_2.*m_p.*r.^2.* ...
    sin(alpha-varphi_p)+l_1.*l_2.*m_w.*r.^2.*sin(alpha-varphi_p).*2.0-l_1.* ...
    m_b.*r.^2.*y_G.*sin(alpha-varphi_p)),(l_1.*l_2.*r.*2.0), ...
    ((I_a.*l_2.^2.*cos(alpha-varphi_p).*2.0+I_b.*r.^2.*cos(alpha-varphi_p) ...
    +I_t.*r.^2.*cos(alpha-varphi_p).*2.0+l_1.^2.*m_b.*r.^2.* ...
    cos(alpha-varphi_p)+l_1.^2.*m_p.*r.^2.*cos(alpha-varphi_p)+ ...
    l_2.^2.*m_w.*r.^2.*cos(alpha-varphi_p).*2.0+m_b.*r.^2.*x_G.^2.* ...
    cos(alpha-varphi_p)+m_b.*r.^2.*y_G.^2.*cos(alpha-varphi_p)-I_a.*l_1.* ...
    l_2.*sin(alpha-varphi_p).*2.0+l_1.*m_p.*r.^2.*x_F.*cos(alpha)-l_1.* ...
    m_p.*r.^2.*y_F.*sin(alpha)+l_1.*m_b.*r.^2.*x_G.*cos(alpha-varphi_p).* ...
    2.0+l_2.*m_b.*r.^2.*y_G.*cos(alpha-varphi_p)-l_1.*l_2.*m_b.*r.^2.* ...
    sin(alpha-varphi_p)-l_1.*l_2.*m_p.*r.^2.*sin(alpha-varphi_p)-l_1.* ...
    l_2.*m_w.*r.^2.*sin(alpha-varphi_p).*2.0-l_1.*m_b.*r.^2.*y_G.* ...
    sin(alpha-varphi_p)).*(-1.0./2.0)./(l_1.*l_2.*r),m_p.* ...
    (x_F.*cos(alpha)-y_F.*sin(alpha)),(r.*(I_p+m_p.*x_F.^2+m_p.*y_F.^2+ ...
    l_1.*m_p.*x_F.*cos(varphi_p)-l_2.*m_p.*y_F.*cos(varphi_p)-l_2.*m_p.* ...
    x_F.*sin(varphi_p)-l_1.*m_p.*y_F.*sin(varphi_p))).*(l_2.*2.0),(r.*(I_p+ ...
    m_p.*x_F.^2+m_p.*y_F.^2+l_1.*m_p.*x_F.*cos(varphi_p)+l_2.*m_p.*y_F.* ...
    cos(varphi_p)+l_2.*m_p.*x_F.*sin(varphi_p)-l_1.*m_p.*y_F.* ...
    sin(varphi_p)).*(-1.0./2.0)./l_2,I_p+m_p.*x_F.^2+m_p.*y_F.^2],[3,3]);

C_bars2 = reshape([-(I_a.*(alpha_dot-varphi_dot_p).*(l_2.*cos(alpha- ...
    varphi_p)+l_1.*sin(alpha-varphi_p))).*(l_1.*r)+(r.*sin(alpha- ...
```

```
    varphi_p).*(alpha_dot-varphi_dot_p).*(l_1.*l_2.*m_w.*-2.0+l_2.*m_b.* ...
    x_G+l_1.*m_b.*y_G))./(l_1.*l_2.*2.0)-(r.*cos(alpha-varphi_p).* ...
    (alpha_dot-varphi_dot_p).*(I_b+I_t.*2.0+l_2.^2.*m_w.*2.0+m_b.*x_G.^2+ ...
    m_b.*y_G.^2+l_1.*m_b.*x_G-l_2.*m_b.*y_G))./(l_1.*l_2.*2.0),(I_a.* ...
    (alpha_dot-varphi_dot_p).*(l_2.*cos(alpha-varphi_p)-l_1.*sin(alpha- ...
    varphi_p)))./(l_1.*r)-(r.*sin(alpha-varphi_p).*(alpha_dot- ...
    varphi_dot_p).*(l_1.*l_2.*m_w.*2.0-l_2.*m_b.*x_G+l_1.*m_b.*y_G))./ ...
    (l_1.*l_2.*2.0)+(r.*cos(alpha-varphi_p).*(alpha_dot-varphi_dot_p).* ...
    (I_b+I_t.*2.0+l_2.^2.*m_w.*2.0+m_b.*x_G.^2+m_b.*y_G.^2+l_1.*m_b.*x_G+ ...
    l_2.*m_b.*y_G))./(l_1.*l_2.*2.0),0.0,(I_a.*(alpha_dot-varphi_dot_p).* ...
    (l_1.*cos(alpha-varphi_p)-l_2.*sin(alpha-varphi_p)))./(l_1.*r)-(r.* ...
    cos(alpha-varphi_p).*(alpha_dot-varphi_dot_p).*(l_1.*l_2.*m_w.*-2.0+ ...
    l_2.*m_b.*x_G+l_1.*m_b.*y_G))./(l_1.*l_2.*2.0)-(r.*sin(alpha- ...
    varphi_p).*(alpha_dot-varphi_dot_p).*(I_b+I_t.*2.0+l_2.^2.*m_w.*2.0+ ...
    m_b.*x_G.^2+m_b.*y_G.^2+l_1.*m_b.*x_G-l_2.*m_b.*y_G))./(l_1.*l_2.* ...
    2.0),(I_a.*(alpha_dot-varphi_dot_p).*(l_1.*cos(alpha-varphi_p)+l_2.* ...
    sin(alpha-varphi_p)))./(l_1.*r)+(r.*cos(alpha-varphi_p).*(alpha_dot- ...
    varphi_dot_p).*(l_1.*l_2.*m_w.*2.0-l_2.*m_b.*x_G+l_1.*m_b.*y_G))./(l_1.* ...
    l_2.*2.0)+(r.*sin(alpha-varphi_p).*(alpha_dot-varphi_dot_p).*(I_b+ ...
    I_t.*2.0+l_2.^2.*m_w.*2.0+m_b.*x_G.^2+m_b.*y_G.^2+l_1.*m_b.*x_G+l_2.* ...
    m_b.*y_G))./(l_1.*l_2.*2.0),0.0,(alpha_dot.*m_p.*r.*(l_2.*x_F.* ...
    cos(varphi_p)+l_1.*y_F.*cos(varphi_p)+l_1.*x_F.*sin(varphi_p)-l_2.* ...
    y_F.*sin(varphi_p)).*(-1.0./2.0))./l_2,(alpha_dot.*m_p.*r.*(-l_2.* ...
    x_F.*cos(varphi_p)+l_1.*y_F.*cos(varphi_p)+l_1.*x_F.*sin(varphi_p)+ ...
    l_2.*y_F.*sin(varphi_p)))./(l_2.*2.0),0.0],[3,3]);

MIIKdots = reshape([-(l_2.*cos(alpha-varphi_p).*(alpha_dot-varphi_dot_p)+ ...
    l_1.*sin(alpha-varphi_p).*(alpha_dot-varphi_dot_p))./(l_1.*r), ...
    (l_2.*cos(alpha-varphi_p).*(alpha_dot-varphi_dot_p)-l_1.*sin(alpha- ...
    varphi_p).*(alpha_dot-varphi_dot_p))./(l_1.*r),(cos(alpha-varphi_p).* ...
    (alpha_dot-varphi_dot_p))./l_1,(l_1.*cos(alpha-varphi_p).*(alpha_dot- ...
    varphi_dot_p)-l_2.*sin(alpha-varphi_p).*(alpha_dot-varphi_dot_p))./ ...
    (l_1.*r),(l_1.*cos(alpha-varphi_p).*(alpha_dot-varphi_dot_p)+l_2.* ...
    sin(alpha-varphi_p).*(alpha_dot-varphi_dot_p))./(l_1.*r),(sin(alpha- ...
    varphi_p).*(alpha_dot-varphi_dot_p))./l_1,0.0,0.0,0.0],[3,3]);

Dmats = reshape([(l_2.*r.*cos(alpha-varphi_p)-l_1.*r.*sin(alpha- ...
    varphi_p))./(l_2.*2.0),(l_1.*r.*cos(alpha-varphi_p)+l_2.*r.* ...
    sin(alpha-varphi_p))./(l_2.*2.0),r./(l_2.*2.0),1.0,0.0,0.0,0.0,(l_2.*r.* ...
    cos(alpha-varphi_p)+l_1.*r.*sin(alpha-varphi_p))./(l_2.*2.0),((l_1.* ...
    r.*cos(alpha-varphi_p)-l_2.*r.*sin(alpha-varphi_p)).*(-1.0./2.0))./ ...
    l_2,(r.*(-1.0./2.0))./l_2,0.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0],[6,3]);

Msys2 = [M_bars2, zeros(3);
         -MIIKs, eye(3)];
```

```
%———————— Compute the equations ————————%

qdot = xs(7:12);
% (!)Warning: given that the u vector from the data is transposed we have to
% transpose it here again that is why in the forula it appears u' insted of
% directly u. We have to take into consideration that in this contex u is a
% 1x3 vector and we want a 3x1.

% Full equations including pivot force disturbances and fricction
% qdotdot = pinv(Msys2)*[u' + Dmats.'*tau_friction + Dmats.'*D_vec - ...
% C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

% Equations including only fricction
qdotdot = pinv(Msys2)*[u' + Dmats.'*tau_friction - C_bars2*qdot(1:3,1); ...
    MIIKdots*qdot(1:3,1)];

% Equations of motion without force disturbances or friction
% qdotdot = pinv(Msys2)*[u' - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

xdot = [qdot; qdotdot];

%———————— Matrices to compute IMU readings ————————%

% Now we configure the necesary matrices
% Rzmat22 = reshape([cos(alpha),sin(alpha),-sin(alpha),cos(alpha)],[2,2]);
IRzmat22 = reshape([cos(alpha)./(cos(alpha).^2+sin(alpha).^2),-sin(alpha)./ ...
    (cos(alpha).^2+sin(alpha).^2),sin(alpha)./(cos(alpha).^2+ ...
    sin(alpha).^2),cos(alpha)./(cos(alpha).^2+sin(alpha).^2)],[2,2]);

% This matrix is the time derivative of the inverse rotation matrix
% (only needed if we use the old way of computing IMU readings)

% dIRzmat22 = reshape([-alpha_dot.*sin(alpha),-alpha_dot.*cos(alpha), ...
% alpha_dot.*cos(alpha),-alpha_dot.*sin(alpha)],[2,2]);

%———————————— Output equations ————————————%

% Old equations of IMU readings (using time derivatives of the inverse
% rotation matrix)
% xyIMU = dIRzmat22*[x_dot;y_dot] + IRzmat22*[qdotdot(1);qdotdot(2)];

% New equations of IMU readings (projection of the acceleration vector to
% another base)
xyIMU = IRzmat22*[qdotdot(1);qdotdot(2)];
```

```
% Output equation using only IMU readings
yout = [xs(9);     % alpha_dot
        xyIMU(1);  % x_dotdot
        xyIMU(2)]; % y_dotdot

end
```

A.3.6 Plot the sampled data

```
%% Plotting the outputs measured and action inputs of the Data File.
% This are the data that we will be feeding to nlgreyest, so it may be
% interesting to plot in order to see exacly what are how are we building
% up the Constrained Minimization Problem.

%———— Plot the outputs ————%

% Plot alphadot coordinate
figure;
plot(data.SamplingInstants, data.OutputData(:,1),'.')
xlabel('t(s)','Interpreter','latex'),ylabel('$\dot{\alpha}$(rad/s)', ...
    'Interpreter','latex'),title('Velocity','Interpreter','latex')

% Plot x_ddot coordinate
figure;
plot(data.SamplingInstants, data.OutputData(:,2),'.')
xlabel('t(s)','Interpreter','latex'),ylabel('$\ddot{x}$(m/$s^2$)', ...
    'Interpreter','latex'),title('Acceleration','Interpreter','latex')

% Plot varphidot_l coordinate
figure;
plot(data.SamplingInstants, data.OutputData(:,3),'.')
xlabel('t(s)','Interpreter','latex'),ylabel('$\ddot{y}$(m/$s^2$)', ...
    'Interpreter','latex'),title('Acceleration','Interpreter','latex')


%———— Plot the inputs ————%

figure;
plot(data.SamplingInstants,data.InputData(:,1:2),'.')
xlabel('t(s)','Interpreter','latex'),ylabel('$\tau$ of wheels(N$\cdot$m)', ...
    'Interpreter','latex'),title('Action Torques','Interpreter','latex')
legend('$\tau$ right','$\tau$ left','Interpreter','latex')

figure;
```

```
plot(data.SamplingInstants,data.InputData(:,3),'.')
xlabel('t(s)','Interpreter','latex'),ylabel('$\tau$ pivot(N$\cdot$m)', ...
    'Interpreter','latex'),title('Action Torques','Interpreter','latex')
```

A.3.7 Compute the errors

```
function [rel_error] = rel_error(real_val,measured_val)
%REL_ERROR Summary of this function goes here
%   This function computes and returns the value of relative error computed
%   as abs(real value - measured value)/ real value * 100. If the real
%   value is 0 then it returns the absolute error without multiplying
lsv = max(size(real_val));

rel_error = zeros(lsv,1);

for i=1:lsv
    if real_val(i,1)~=0
        rel_error(i,1) = abs(real_val(i,1) - measured_val(i,1))./ ...
            abs(real_val(i,1))*100;
    else
        rel_error(i,1) = abs(real_val(i,1) - measured_val(i,1));
    end
end
end
```

A.3.8 Save results group of functions

```
%% This script is made to save the excel results tables

% Compute the % of deviation

percent = TSinputDev;
percent = num2str(percent,'%.0f');

fullsavename = strcat(FullDirNameExcel,percent,'%','.xlsx');

% witing the table
writetable(ResultsTable,fullsavename)
```

```
%% This script is made to save the Results Report given by nlgreyest

filename = 'report_file';

ReportNLGRM = nlgrm.Report; % Create report object

percent = TSinputDev;
```

```
percent = num2str(percent,'%.0f');
Tsvalue = num2str(data.Ts,'%.3f');

fullsavename = strcat(PartialDirNameExcel,'/',filename,Tsvalue,'s_', ...
    percent,'.mat');

save(fullsavename,ReportNLGRM)
```

```
%% Write a txt file with the termination condition
docname = 'term_cond';

TCnlgrm = ReportNLGRM.Termination;
OpUnlgrm = ReportNLGRM.OptionsUsed;

fulldocname = strcat(PartialDirNameExcel,'/',docname,Tsvalue,'s_', ...
    percent,'.txt');

fileID = fopen(fulldocname,'w');
fprintf(fileID,'%s %s\r\n','Method: ',ReportNLGRM.Method);
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s\r\n','Termination Report');
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s %s\r\n','WhyStop: ',TCnlgrm.WhyStop);
fprintf(fileID,'%s %.0f\r\n','Iterations: ',TCnlgrm.Iterations);
fprintf(fileID,'%s %.4e\r\n','FirstOrderOptimality: ', ...
    TCnlgrm.FirstOrderOptimality);
fprintf(fileID,'%s %.0f\r\n','FcnCount: ',TCnlgrm.FcnCount);
fprintf(fileID,'%s %s\r\n','Algorithm: ',TCnlgrm.Algorithm);
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s\r\n','OptionsUsed Report');
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s %s\r\n','SearchMethod: ',OpUnlgrm.SearchMethod);
fprintf(fileID,'%s %.0f\r\n','MaxIterations: ', ...
    OpUnlgrm.SearchOptions.MaxIterations);
fprintf(fileID,'%s %.4e\r\n','StepTolerance: ', ...
    OpUnlgrm.SearchOptions.StepTolerance);
fprintf(fileID,'%s %.4e\r\n','FunctionTolerance: ', ...
    OpUnlgrm.SearchOptions.FunctionTolerance);
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s\r\n','Time to solve Report');
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s %.4f\r\n','Elapsed time [s]: ',tEnd);
fclose(fileID);
```

```
%% This script is made to save the Results Report given by nlgreyest

filename = 'report_file';

ReportNLGRM = nlgrm.Report; % Create report object

percent = TSinputDev;

percent = num2str(percent,'%.0f');
Tsvalue = num2str(data.Ts,'%.3f');

fullsavename = strcat(PartialDirNameExcel,'/',filename,Tsvalue,'s_', ...
    percent,'.mat');

save(fullsavename,ReportNLGRM)
```

```
%% Write a txt file with the termination condition
docname = 'term_cond';

TCnlgrm = ReportNLGRM.Termination;
OpUnlgrm = ReportNLGRM.OptionsUsed;

fulldocname = strcat(PartialDirNameExcel,'/',docname,Tsvalue,'s_', ...
    percent,'.txt');

fileID = fopen(fulldocname,'w');
fprintf(fileID,'%s %s\r\n','Method: ',ReportNLGRM.Method);
fprintf(fileID,'%s\r\n','————————————————————————————');
fprintf(fileID,'%s\r\n','Termination Report');
fprintf(fileID,'%s\r\n','————————————————————————————');
fprintf(fileID,'%s %s\r\n','WhyStop: ',TCnlgrm.WhyStop);
fprintf(fileID,'%s %.0f\r\n','Iterations: ',TCnlgrm.Iterations);
fprintf(fileID,'%s %.4e\r\n','FirstOrderOptimality: ', ...
    TCnlgrm.FirstOrderOptimality);
fprintf(fileID,'%s %.0f\r\n','FcnCount: ',TCnlgrm.FcnCount);
fprintf(fileID,'%s %s\r\n','Algorithm: ',TCnlgrm.Algorithm);
fprintf(fileID,'%s\r\n','————————————————————————————');
fprintf(fileID,'%s\r\n','OptionsUsed Report');
fprintf(fileID,'%s\r\n','————————————————————————————');
fprintf(fileID,'%s %s\r\n','SearchMethod: ',OpUnlgrm.SearchMethod);
fprintf(fileID,'%s %.0f\r\n','MaxIterations: ', ...
    OpUnlgrm.SearchOptions.MaxIterations);
fprintf(fileID,'%s %.4e\r\n','StepTolerance: ', ...
    OpUnlgrm.SearchOptions.StepTolerance);
fprintf(fileID,'%s %.4e\r\n','FunctionTolerance: ', ...
```

```
    OpUnlgrm.SearchOptions.FunctionTolerance);
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s\r\n','Time to solve Report');
fprintf(fileID,'%s\r\n','————————————————————————————————');
fprintf(fileID,'%s %.4f\r\n','Elapsed time [s]: ',tEnd);
fclose(fileID);
```

```
%% Seccio 1
NameExistence = exist('CN1_SP', 'var');

if NameExistence == 0
    CN = '  sampled_data_plots.png'; % Name the image
else
    CN = CN1_SP;
end

Direxistance = exist('FullDirNameSP','var');

if Direxistance == 0
    for i=1:5
        if i==1
            names='1 - alphadot';
        elseif i==2
            names='2 - x_ddot';
        elseif i==3
            names='3 - y_ddot';
        elseif i==4
            names='4 - taus wheels';
        elseif i==5
            names='5 - tau pivot';
        end
        saveas(figure(i),strcat(names,CN));
    end
else
    for i=1:5
        if i==1
            names='1 - alphadot';
        elseif i==2
            names='2 - x_ddot';
        elseif i==3
            names='3 - y_ddot';
        elseif i==4
            names='4 - taus wheels';
        elseif i==5
            names='5 - tau pivot';
```

```
        end
        saveas(figure(i),strcat(FullDirNameSP,'/',names,CN));
    end
end
```

```
%% This script is made to save the required data to make the sensitivity plots

filename = 'sensitivity_data';

% Compute the % of deviation
Dev_percentage = com_ini_dev(ParameterRealValues, ParameterInitialGuesses, ...
    TSinputDev);

% Setting up the Free or fixed parameters for estimation vector

FixedVector = [nlgrm.parameters(1).Fixed;   % I_b [kg*m^2]
               nlgrm.parameters(2).Fixed;  % I_p [kg*m^2]
               nlgrm.parameters(3).Fixed;  % I_a [kg*m^2]
               nlgrm.parameters(4).Fixed;  % I_t [kg*m^2]
               nlgrm.parameters(5).Fixed;  % l_1 [m]
               nlgrm.parameters(6).Fixed;  % l_2 [m]
               nlgrm.parameters(7).Fixed;  % m_b [kg]
               nlgrm.parameters(8).Fixed;  % m_w [kg]
               nlgrm.parameters(9).Fixed;  % m_p [kg]
               nlgrm.parameters(10).Fixed; % x_G [m]
               nlgrm.parameters(11).Fixed; % y_G [m]
               nlgrm.parameters(12).Fixed; % x_F [m]
               nlgrm.parameters(13).Fixed; % y_F [m]
               nlgrm.parameters(14).Fixed; % r   [m]
               nlgrm.parameters(15).Fixed; % b_r [kg*m^2*s^-1]
               nlgrm.parameters(16).Fixed; % b_l [kg*m^2*s^-1]
               nlgrm.parameters(17).Fixed];% b_p [kg*m^2*s^-1]
%% Saveing all the Data in a .mat file

percent = TSinputDev;
percent = num2str(percent,'%.0f');
Tsvalue = num2str(data.Ts,'%.3f');

fullsavename = strcat('SensitivityData/',filename,Tsvalue,'s_',percent,'.mat');

save(fullsavename,'Dev_percentage','FixedVector','RelativeError_Percentage')

%% Extra functions in this document

function Dev_C = com_ini_dev(RVal, IGuess, TSinputDev)
```

```
lsv = max(size(RVal));

Dev_C = zeros(lsv,1);

for i=1:lsv
    if RVal(i,1)~=0
        Dev_C(i,1) = abs(RVal(i,1) - IGuess(i,1))./abs(RVal(i,1))*100;
    else
        Dev_C(i,1) = TSinputDev; % We give the value inputed directly
    end
end

end
```

```
%% This script is made to save the final values of the estimated parameters

filename = 'final_pars';

percent = TSinputDev;
percent = num2str(percent,'%.0f');
Tsvalue = num2str(data.Ts,'%.3f');

fullsavename = strcat(PartialDirNameExcel,'/',filename,Tsvalue,'s_', ...
    percent,'.mat');
fullsavename2 = strcat('Final_Pars_Sim/Otbot1_CTC_W1_DF_IMU/SetParameters/', ...
    filename,'.mat');

save(fullsavename,ParameterFinalValues)
save(fullsavename2,ParameterFinalValues)
```

```
%% Now we will be saveing the Tsvalue, inputDeviation and Tsaction

Tsvalnum = data.Ts;
save('Final_Pars_Sim/Otbot1_CTC_W1_DF_IMU/a0_Tsvalnum.mat',Tsvalnum)
save('Final_Pars_Sim/Otbot1_CTC_W1_DF_IMU/a1_inputDev.mat',TSinputDev)

% Save the time of action holding haction or Tsaction
prompt = 'Introduce the time value of holding action, haction or' ...
    ' Tsaction [0.01]: ';
Tsaction = input(prompt);
if isempty(Tsaction)
    Tsaction = 0.01;
end
```

```
save('Final_Pars_Sim/Otbot1_CTC_W1_DF_IMU/a3_Tsaction.mat',Tsaction)

% Save the time of simulation
prompt = 'Introduce total time of simulation [1]: ';
Tsim = input(prompt);
if isempty(Tsim)
    Tsim = 1;
end

save('Final_Pars_Sim/Otbot1_CTC_W1_DF_IMU/a4_Tsim.mat',Tsim)

% Saveing Sampling instants
SampInst = data.SamplingInstants;
save('Final_Pars_Sim/Otbot1_CTC_W1_DF_IMU/a5_SampInstants.mat',SampInst)
```

A.4 Control routines

The scheme followed to achieve the simulations with control is the same as the one used in the first part of the dynamic simulation, in Section A.2. It is only necessary to incorporate some changes in the main script `Main_Simulation.m`, and in the function containing the dynamic equations `xdot_Otbot.m`, so that they include the control law. We also use a particular function here called `build_cntrl.m` to construct the matrices $\mathbf{A}$ and $\mathbf{B}$ of the linear model, along with the list of poles or eigenvalues to adjust the control law and therefore the gain matrices $\mathbf{K}_p$ and $\mathbf{K}_v$ concatenated as a single one $\mathbf{K} = [\mathbf{K}_p, \mathbf{K}_v]$. Finally, we also have two functions `TD_circle_of_time_XYA.m`, `TD_polyline_of_time.m` to compute the trajectories to be fed to the controller. The code used to do this is shown below.

A.4.1 Main simulation with control

```
clearvars
close all
clc
%% Flags
% Flag to define if you want video output or not
FlagVideo = 'YES';
               % (FlagVideo == YES)> simulation with VIDEO output
               % (FlagVideo == NO )> simulation without VIDEO output

% Flag to define if you want a constnat torque for all simulation or not
u_Flag = 'CTC_PP';
               % (u_Flag == CTE)> simulation with constant torques
               % (u_Flag == VAR)> simulation with a torque function of time
               % (u_Flag == CTC_PP)> simulation with a CTC controler using
               % pole placement method
```

```
                % Now we create torques comand vector
                % (Only used if u_Flag is set to CTE or VAR)
                u = [0.1;0.1;-0.1]; % All in Nm

% Flag to define the desired trajectory for the robot, only works if u is set
% to PP
goal_Flag = 'CIRCLE';
               % (goal_Flag == FIX)> goal trajectory is a fixed point
               % (goal_Flag == CIRCLE)> goal trajectory is a circle
               % (goal_Flag == POLILINE)> goal trajectory is a poliline

               % Set a fixed point in case goal_Flag = FIX
               % [x,y,alpha,varphi_r,varphi_l,varphi_p] velocities are 0
                Fix_point = [2,2,0,0,0,0]';

% Flag to define if we want to add friction in our model or not
FrictFlag = 'NO';
               % (FrictFlag == YES)> Our model will have frictions
               % (FrictFlag == NO)> Our model will not contain frictions

% Flag to define if we want to add non modelled disturbances during the
% simulation
DistFlag = 'NO';
               % (DistFlag == YES)> Non modelled disturbances will be added
               % (DistFlag == NO)> Non modelled disturbances will NOT be added

% Flag to define if you want to track the path of each center of mass
TrackFlag = 'PF';
               % (TrackFlag == NO)> plots without any paths of center of
               % mass plotted

               % (TrackFlag == CB)> plots the path of the center of mass of
               % the chassis body

               % (TrackFlag == PF)> plots the path of the center of mass of
               % the platform body

               % (TrackFlag == BOTH)> plots the path of both centers of
               % mass

% Flag to choose if you want to see the desired goal in the video simulation
% (Only avaliable if u_Flag is CTC_PP)
TargetFlag = 'YES';
                % (TargetFlag == YES)> A red target will be displayed in the
                % simulation
```

```
                % (TargetFlag == NO)> There won't be a target in the video
                % simulation

% Flag to choose if you want to compute and display the plots of action vector u
UvecFlag = 'YES';
                % (UvecFlag == YES)> action vector u plots will be displayed
                % (UvecFlag == NO)> action vector u plots will not be displayed

KinEnFlag = 'NO'; % Flag to choose if Kinetic energic plot must be displayed or
                  % not
               % (KinEnFlag == YES)> Outputs a plot of kinetic energy
               % (KinEnFlag == NO )> No output plot of kinetic energy

Jdot_qdotFlag = 'NO'; % Flag to choose if plot of equation Jdot*qdot is equal
                      % to 0 during all simulation
               % (Jdot_qdotFlag == YES)> Outputs a plot of this equation
               % (Jdot_qdotFlag == NO)> No output of this plot

ErrorFlag = 'YES'; % Flag to choose if error plots are displayed or not
               % (ErrorFlag == YES)> Error plots will be displayed
               % (ErrorFlag == NO)> Error plots will not be displayed

%% Setting up simulation parameters

tf = 15;      % Final simulation time
h = 0.001;    % Number of samples within final vectors (times and states)
opts = odeset('MaxStep',1e-3); % Max integration time step

%% Setting up model parameters & matrices
load('m_struc')
load('sm_struc')

%% Initial conditions
p0 = [0,0,0]';
varphi0 = [0,0,0]';

pdot0 = zeros(3,1);

% Need MIIK
MIIKmat = sm.MIIKmatrix(p0(3),varphi0(3));
% Computing p0
varphidot0 = MIIKmat*pdot0;

% Initial conditions for the system
xs0=[p0; varphi0; pdot0; varphidot0];
```

```
%% Desired trajectory
if strcmp(u_Flag,'CTC_PP')
    if strcmp(goal_Flag,'FIX')
        cp.pss = Fix_point;
    else
        cp.pss = zeros(6,1);
    end
end
% cp struct object stands for control parameters in contains the desires
% goal fix point and the gain matrix

%% Build the CTC controller
% define the set of poles or eigenvalues
EIG_set = [-4/3, -4/3, -4/3, -40/3, -40/3, -40/3];
cp.K = build_cntrl(EIG_set);

%% Building the diferential equation
dxdt= @(t,xs) xdot_otbot(t, xs, m, sm, u, u_Flag, cp, goal_Flag, ...
    FrictFlag, DistFlag);

%% Simulationg with ode45
t = 0:h:tf;
[times,states]=ode45(dxdt,t,xs0,opts);

%% Vectorize the path of centers of mass
if strcmp(TrackFlag,'NO')==0

    % Setting this to adjust it to the video, here and down below they must
    % heve the same values, n & fs
    fs=30;
    n=round(1/(fs*h));

    % Set, ploting factor:
    % This plot factor is to scale if we want compute the com of each body every
    % frame (Gpf = 1) or less frequently i.e every 2 frames (Gpf = 2...)
    Gpf = 16;
    n2 = Gpf*n;

    aux1=length(times);
    GBpmat = zeros(3,round(aux1/n2));
    GPpmat = zeros(3,round(aux1/n2));
    jaux=1;

    for i = 1:n2:aux1
```

```
        q.x = states(i,1);
        q.y = states(i,2);
        q.alpha = states(i,3);
        q.varphi_r = states(i,4);
        q.varphi_l = states(i,5);
        q.varphi_p = states(i,6);

        [GBpi,GPpi] = c_o_m_bodies(m,q);
        GBpmat(:,jaux) = GBpi;
        GPpmat(:,jaux) = GPpi;
        jaux=jaux+1;
    end
end
```

```
%% Compute KinEnergy in Every instant
if strcmp(KinEnFlag,'YES')
    ls = length(states);
    kinenvalues = zeros(ls,1);
   for i=1:ls
       kinenvalues(i,1) = sm.Texpr(states(i,3),states(i,9),states(i,6), ...
           states(i,11),states(i,12),states(i,10),states(i,7),states(i,8));
   end
end
```

```
%% Compute the error for every instant
uswitch = strcmp(u_Flag,'CTC_PP');

if strcmp(ErrorFlag,'YES') && uswitch
    lst = length(states);
    errorstates = zeros(lst,6);
    switch goal_Flag
        case 'FIX'
            for i = 1:lst
                errorstates(i,1:3) = cp.pss(1:3)' - states(i,1:3);
                errorstates(i,4:6) = cp.pss(4:6)' - states(i,7:9);
            end
        case 'CIRCLE'
            for i=1:lst
                [Xc,Yc,Alphac,Xdotc,Ydotc,Alphadotc,Xddotc,Yddotc,Alphaddotc] ...
                    = TD_circle_of_time_XYA(times(i));
                errorstates(i,1:3) = [Xc,Yc,Alphac] - states(i,1:3);
                errorstates(i,4:6) = [Xdotc,Ydotc,Alphadotc] - states(i,7:9);
            end
        case 'POLILINE'
            for i=1:lst
```

```
                [Xc,Yc,Alphac,Xdotc,Ydotc,Alphadotc,Xddotc,Yddotc,Alphaddotc] ...
                    = TD_polyline_of_time(times(i));
                errorstates(i,1:3) = [Xc,Yc,Alphac] - states(i,1:3);
                errorstates(i,4:6) = [Xdotc,Ydotc,Alphadotc] - states(i,7:9);
            end

    end
end
```

```
%% Compute desired GOAL (Target) in Every instant

uswitch = strcmp(u_Flag,'CTC_PP');

if strcmp(TargetFlag,'YES') && strcmp(FlagVideo,'YES') && uswitch
    TEFlag = 0;
    lst = length(states);
    switch goal_Flag
        case 'FIX'
            targetpos = repmat(cp.pss',[lst,1]);

        case 'CIRCLE'
            targetpos = zeros(lst,3);
            for i=1:lst
                [Xc,Yc,Alphac,aux1,aux2,aux3,aux4,aux5,aux6] ...
                    = TD_circle_of_time_XYA(times(i));
                targetpos(i,:) = [Xc,Yc,Alphac];
            end
        case 'POLILINE'
            targetpos = zeros(lst,3);
            for i=1:lst
                [Xc,Yc,Alphac,aux1,aux2,aux3,aux4,aux5,aux6] ...
                    = TD_polyline_of_time(times(i));
                targetpos(i,:) = [Xc,Yc,Alphac];
            end
        otherwise
            disp('This goal_Flag does not exist cant compute the Target')
            TEFlag = 1;
    end
else
    TEFlag = 1;
    disp('The combination of flags used is not correct to compute Target')
    disp('Please change the flags to a possible combination in order to' ...
    'display the desired results')
    disp('Target loaction will not be computed')
end
```

```
%% Compute action torques in every instant
if strcmp(UvecFlag,'YES')
    u_vector = zeros(tf/h+1,4);
    switch u_Flag
        case 'CTE'
            ls = length(states);
            for i=1:ls
                u_vector(i,1:4) = [u',times(i)];
            end
        case 'VAR'
            ls = length(states);
            for i=1:ls
                u_f = u_function(times(i),u);
                u_vector(i,1:4) = [u_f',times(i)];
            end
         case 'CTC_PP'
            switch goal_Flag
                case 'FIX'
                    ls = length(states);
                    for i=1:ls
                        M_barsout2 = sm.M_barmatrix2(states(i,3),states(i,6));
                        C_barsout2 = sm.C_barmatrix2(states(i,3),states(i,9), ...
                            states(i,6),states(i,12));

                        pdiffout(1:3,1) = cp.pss(1:3,1) - states(i,1:3)';
                        pdiffout(4:6,1) = cp.pss(4:6,1) - states(i,7:9)';

                        u_2out = cp.K*pdiffout;
                        uout = M_barsout2*u_2out + C_barsout2*states(i,7:9)';
                        u_vector(i,1:4) = [uout',times(i)];
                    end
                case 'CIRCLE'
                    ls = length(states);
                    for i=1:ls
                        M_barsout2 = sm.M_barmatrix2(states(i,3),states(i,6));
                        C_barsout2 = sm.C_barmatrix2(states(i,3),states(i,9), ...
                        states(i,6),states(i,12));

                        [xssout,yssout,alphassout,xdotssout,ydotssout, ...
                            alphadotssout,xdotdotssout,ydotdotssout, ...
                            alphadotdotssout] = TD_circle_of_time_XYA(times(i));
                        pssout = [xssout;yssout;alphassout;xdotssout; ...
                            ydotssout;alphadotssout];
                        pd_dotdotout = [xdotdotssout; ydotdotssout; ...
```

```
                alphadotdotssout];

            pdiffout(1:3,1) = pssout(1:3,1) - states(i,1:3)';
            pdiffout(4:6,1) = pssout(4:6,1) - states(i,7:9)';

            u_2out = pd_dotdotout + cp.K*pdiffout;
            uout = M_barsout2*u_2out + C_barsout2*states(i,7:9)';
            u_vector(i,1:4) = [uout',times(i)];
        end
    case 'POLILINE'
        ls = length(states);
        for i=1:ls
            M_barsout2 = sm.M_barmatrix2(states(i,3),states(i,6));
            C_barsout2 = sm.C_barmatrix2(states(i,3),states(i,9), ...
                states(i,6),states(i,12));

            [xssout,yssout,alphassout,xdotssout,ydotssout, ...
                alphadotssout,xdotdotssout,ydotdotssout, ...
                alphadotdotssout] = TD_polyline_of_time(times(i));
            pssout = [xssout;yssout;alphassout;xdotssout; ...
                ydotssout;alphadotssout];
            pd_dotdotout = [xdotdotssout; ydotdotssout; ...
                alphadotdotssout];

            pdiffout(1:3,1) = pssout(1:3,1) - states(i,1:3)';
            pdiffout(4:6,1) = pssout(4:6,1) - states(i,7:9)';

            u_2out = pd_dotdotout + cp.K*pdiffout;
            uout = M_barsout2*u_2out + C_barsout2*states(i,7:9)';
            u_vector(i,1:4) = [uout',times(i)];
        end
    otherwise
        disp('This goal_Flag does not exist for the torques' ...
            'computation')
        disp('Simulating with a FIX point X=2 Y=2 Alpha = 0' ...
            'with standart PD CTC')
        cp.pss = [2,2,0,0,0,0]';
        ls = length(states);
        for i=1:ls
            M_barsout2 = sm.M_barmatrix2(states(i,3),states(i,6));
            C_barsout2 = sm.C_barmatrix2(states(i,3),states(i,9), ...
                states(i,6),states(i,12));

            pdiffout(1:3,1) = cp.pss(1:3,1) - states(i,1:3)';
            pdiffout(4:6,1) = cp.pss(4:6,1) - states(i,7:9)';
```

```
                        u_2out = cp.K*pdiffout;

                        uout = M_barsout2*u_2out + C_barsout2*states(i,7:9)';
                        u_vector(i,1:4) = [uout',times(i)];
                    end
            end
        otherwise
            disp('Warning this u_Flag does not exits omiting torques' ...
                'camputation')
            disp('Computation with CTE torques will be launched')
            ls = length(states);
            for i=1:ls
                u_vector(i,1:4) = [u',times(i)];
            end
    end
else
    disp('Plots of the evolution of actions u will not be displayed')
end
```

```
%% Plot results
figure;
plot(times,states(:,1))
xlabel('t(s)','Interpreter','latex'),ylabel('$x$(meters)','Interpreter', ...
    'latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,2))
xlabel('t(s)','Interpreter','latex'),ylabel('$y$(meters)','Interpreter', ...
    'latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,3))
xlabel('t(s)','Interpreter','latex'),ylabel('$\alpha$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,4))
xlabel('t(s)','Interpreter','latex'),ylabel('$\varphi_r$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,5))
xlabel('t(s)','Interpreter','latex'),ylabel('$\varphi_l$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')
```

```
figure;
plot(times,states(:,6))
xlabel('t(s)','Interpreter','latex'),ylabel('$\varphi_p$(radians)', ...
    'Interpreter','latex'),title('Configuration','Interpreter','latex')

figure;
plot(times,states(:,7))
xlabel('t(s)','Interpreter','latex'),ylabel('$\dot{x}$(meters/second)', ...
    'Interpreter','latex'),title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,8))
xlabel('t(s)','Interpreter','latex'),ylabel('$\dot{y}$(meters/second)', ...
    'Interpreter','latex'),title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,9))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\alpha}$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,10))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\varphi}_r$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,11))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\varphi}_l$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

figure;
plot(times,states(:,12))
xlabel('t(s)','Interpreter','latex'), ...
ylabel('$\dot{\varphi}_p$(radians/second)','Interpreter','latex'), ...
title('Velocity','Interpreter','latex')

% Plot the action vectors
if strcmp(UvecFlag,'YES')
    figure;
    plot(times,u_vector(:,1:2))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\tau$ of wheels(N$\cdot$m)','Interpreter','latex'), ...
```

```
    title('Action Torques','Interpreter','latex')
    legend('$\tau$ right','$\tau$ left','Interpreter','latex')

    figure;
    plot(times,u_vector(:,3))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\tau$ pivot(N$\cdot$m)','Interpreter','latex'), ...
    title('Action Torques','Interpreter','latex')
end

if strcmp(KinEnFlag,'YES')
    figure;
    plot(times,kinenvalues)
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('Kinetic Energy (Joules)','Interpreter','latex'), ...
    title('System Energy','Interpreter','latex')
end

if strcmp(ErrorFlag,'YES') && uswitch
    figure;
    plot(times,errorstates(:,1))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$x$ error(meters)','Interpreter','latex'), ...
    title('Configuration Error','Interpreter','latex')

    figure;
    plot(times,errorstates(:,2))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$y$ error(meters)','Interpreter','latex'), ...
    title('Configuration Error','Interpreter','latex')

    figure;
    plot(times,errorstates(:,3))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\alpha$ error(radians)','Interpreter','latex'), ...
    title('Configuration Error','Interpreter','latex')

    figure;
    plot(times,errorstates(:,4))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\dot{x}$ error(meters/s)','Interpreter','latex'), ...
    title('Velocity Error','Interpreter','latex')

    figure;
    plot(times,errorstates(:,5))
```

```
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\dot{y}$ error(meters/s)','Interpreter','latex'), ...
    title('Velocity Error','Interpreter','latex')

    figure;
    plot(times,errorstates(:,6))
    xlabel('t(s)','Interpreter','latex'), ...
    ylabel('$\dot{\alpha}$ error(radians/s)','Interpreter','latex'), ...
    title('Velocity Error','Interpreter','latex')
else
    disp('Error plots will not be displayed')
end
```

```
%% Seting Jdot_qdot = 0 Flag

if strcmp(Jdot_qdotFlag,'YES')
    ls = length(states);
    J_qvalues = zeros(ls,3);
    for i=1:ls
        Jplot = sm.Jmatrix(states(i,3),states(i,6));
        qdotplot = states(i,7:12)';
        J_qvalues(i,:) = (Jplot*qdotplot)';
    end
    figure;
    plot(times,J_qvalues)
    xlabel('t(s)'),ylabel('Result vector'), title('Vector result of J*qdot')
end
```

```
%% Movie and video of the simulation
switch FlagVideo
    case 'YES'
        switch TrackFlag
            case 'NO'
                time=cputime;
                % Animation
                % h is the sampling time
                % n is the scaling factor in order not to plot with the same step
                % than during the integration with ode45

                fs=30;
                n=round(1/(fs*h));

                % Set up the movie.
                writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
                %writerObj.FileFormat = 'mp4';
```

```
writerObj.FrameRate = fs; % How many frames per second.
open(writerObj);

TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
if strcmp(TargetFlag,'NO') || TEFlagcheck
    for i = 1:n:length(times)
            q.x = states(i,1);
            q.y = states(i,2);
            q.alpha = states(i,3);
            q.varphi_r = states(i,4);
            q.varphi_l = states(i,5);
            q.varphi_p = states(i,6);

            elapsed = cputime-time;
            if elapsed > 1200
                disp(elapsed);
                disp('took too long to generate the video')
                break
            else
                draw_otbot(m,q) % Drawing frame
                hold on
                if strcmp(DistFlag,'YES') && ...
                        norm(Dist_funct(times(i))) ~= 0
                    Dist_video = Dist_funct(times(i));
                    D_XY = Dist_video(1:2,1);
                    D_XY = D_XY./norm(D_XY);
                    D_XY = 3*m.l_1*D_XY;
                    PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                        D_XY(1,1)));
                    arrows(states(i,1) - D_XY(1,1), ...
                        states(i,2) - D_XY(2,1), norm(D_XY), ...
                        270 - PolAng , 'FaceColor','r', ...
                        'EdgeColor','none');
                end
                hold off

                % 'gcf' can handle if you zoom in to take a movie
                    .
                frame = getframe(gcf);
                writeVideo(writerObj, frame);
            end

    end
    close(writerObj); % Saves the movie.
```

```
elseif strcmp(TargetFlag,'YES') && TEFlag == 0
    for i = 1:n:length(times)
            q.x = states(i,1);
            q.y = states(i,2);
            q.alpha = states(i,3);
            q.varphi_r = states(i,4);
            q.varphi_l = states(i,5);
            q.varphi_p = states(i,6);

            elapsed = cputime-time;
            if elapsed > 1200
                disp(elapsed);
                disp('took too long to generate the video')
                break
            else
                draw_otbot(m,q)
                hold on
                otbot_circle_v2(targetpos(i,1), ...
                    targetpos(i,2),m.l_2/4,'r');
                hold off

                hold on
                if strcmp(DistFlag,'YES') && ...
                        norm(Dist_funct(times(i))) ~= 0
                    Dist_video = Dist_funct(times(i));
                    D_XY = Dist_video(1:2,1);
                    D_XY = D_XY./norm(D_XY);
                    D_XY = 3*m.l_1*D_XY;
                    PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                        D_XY(1,1)));
                    arrows(states(i,1) - D_XY(1,1), ...
                        states(i,2) - D_XY(2,1), norm(D_XY), ...
                        270 - PolAng , 'FaceColor','r', ...
                        'EdgeColor','none');
                end
                hold off

                % 'gcf' can handle if you zoom in to take a movie
                    .
                frame = getframe(gcf);
                writeVideo(writerObj, frame);
            end

    end
    close(writerObj); % Saves the movie.
```

```
        end

    case 'BOTH'
        time=cputime;
        % Animation
        % h is the sampling time
        % n is the scaling factor in order not to plot with the same step
        % than during the integration with ode45

        fs=30;
        n=round(1/(fs*h));

        % Set up the movie.
        writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
        %writerObj.FileFormat = 'mp4';
        writerObj.FrameRate = fs; % How many frames per second.
        open(writerObj);
        jaux = 1;
        jaux2 = 1;

        TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
        if strcmp(TargetFlag,'NO') || TEFlagcheck
            for i = 1:n:length(times)
                    q.x = states(i,1);
                    q.y = states(i,2);
                    q.alpha = states(i,3);
                    q.varphi_r = states(i,4);
                    q.varphi_l = states(i,5);
                    q.varphi_p = states(i,6);

                    elapsed = cputime-time;
                    if elapsed > 1200
                        disp(elapsed);
                        disp('took too long to generate the video')
                        break
                    else
                        draw_otbot(m,q)
                        if jaux2 == 1
                            GBpplot = GBpmat(:,1:jaux);
                            GPpplot = GPpmat(:,1:jaux);
                            multi_circles(GBpplot,m.l_2/6,'b')
                            hold on
                            multi_circles(GPpplot,m.l_2/6,'g')
                            jaux2 = jaux2 + 1;
```

```
                    jaux = jaux+1;
                elseif jaux2>= Gpf
                    multi_circles(GBpplot,m.l_2/6,'b')
                    hold on
                    multi_circles(GPpplot,m.l_2/6,'g')
                    jaux2 = 1;
                else
                    multi_circles(GBpplot,m.l_2/6,'b')
                    hold on
                    multi_circles(GPpplot,m.l_2/6,'g')
                    jaux2 = jaux2 + 1;
                end

                hold on
                if strcmp(DistFlag,'YES') && ...
                        norm(Dist_funct(times(i))) ~= 0
                    Dist_video = Dist_funct(times(i));
                    D_XY = Dist_video(1:2,1);
                    D_XY = D_XY./norm(D_XY);
                    D_XY = 3*m.l_1*D_XY;
                    PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                        D_XY(1,1)));
                    arrows(states(i,1) - D_XY(1,1), ...
                        states(i,2) - D_XY(2,1), norm(D_XY), ...
                        270 - PolAng , 'FaceColor','r', ...
                        'EdgeColor','none');
                end
                hold off

                % 'gcf' can handle if you zoom in to take a movie
                    .
                frame = getframe(gcf);
                writeVideo(writerObj, frame);
            end
        end
        close(writerObj); % Saves the movie.

    elseif strcmp(TargetFlag,'YES') && TEFlag == 0
        for i = 1:n:length(times)
                q.x = states(i,1);
                q.y = states(i,2);
                q.alpha = states(i,3);
                q.varphi_r = states(i,4);
                q.varphi_l = states(i,5);
                q.varphi_p = states(i,6);
```

```
elapsed = cputime-time;
if elapsed > 1200
    disp(elapsed);
    disp('took too long to generate the video')
    break
else
    draw_otbot(m,q)
    if jaux2 == 1
        GBpplot = GBpmat(:,1:jaux);
        GPpplot = GPpmat(:,1:jaux);
        multi_circles(GBpplot,m.l_2/6,'b')
        hold on
        multi_circles(GPpplot,m.l_2/6,'g')
        hold on
        otbot_circle_v2(targetpos(i,1), ...
            targetpos(i,2),m.l_2/4,'r');
        hold off
        jaux2 = jaux2 + 1;
        jaux = jaux+1;
    elseif jaux2>= Gpf
        multi_circles(GBpplot,m.l_2/6,'b')
        hold on
        multi_circles(GPpplot,m.l_2/6,'g')
        hold on
        otbot_circle_v2(targetpos(i,1), ...
            targetpos(i,2),m.l_2/4,'r');
        hold off
        jaux2 = 1;
    else
        multi_circles(GBpplot,m.l_2/6,'b')
        hold on
        multi_circles(GPpplot,m.l_2/6,'g')
        hold on
        otbot_circle_v2(targetpos(i,1), ...
            targetpos(i,2),m.l_2/4,'r');
        hold off
        jaux2 = jaux2 + 1;
    end

    hold on
    if strcmp(DistFlag,'YES') && ...
            norm(Dist_funct(times(i))) ~= 0
        Dist_video = Dist_funct(times(i));
        D_XY = Dist_video(1:2,1);
```

```
                            D_XY = D_XY./norm(D_XY);
                            D_XY = 3*m.l_1*D_XY;
                            PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                                D_XY(1,1)));
                            arrows(states(i,1) - D_XY(1,1), ...
                                states(i,2) - D_XY(2,1), norm(D_XY), ...
                                270 - PolAng , 'FaceColor','r', ...
                                'EdgeColor','none');
                        end
                        hold off

                        % 'gcf' can handle if you zoom in to take a movie
                            .
                        frame = getframe(gcf);
                        writeVideo(writerObj, frame);
                    end
                end
                close(writerObj); % Saves the movie.

            end

        case 'CB'
            time=cputime;
            % Animation
            % h is the sampling time
            % n is the scaling factor in order not to plot with the same step
            % than during the integration with ode45

            fs=30;
            n=round(1/(fs*h));

            % Set up the movie.
            writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
            %writerObj.FileFormat = 'mp4';
            writerObj.FrameRate = fs; % How many frames per second.
            open(writerObj);
            jaux = 1;
            jaux2 = 1;

            TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
            if strcmp(TargetFlag,'NO') || TEFlagcheck
                for i = 1:n:length(times)
                        q.x = states(i,1);
                        q.y = states(i,2);
                        q.alpha = states(i,3);
```

```
q.varphi_r = states(i,4);
q.varphi_l = states(i,5);
q.varphi_p = states(i,6);

elapsed = cputime-time;
if elapsed > 1200
    disp(elapsed);
    disp('took too long to generate the video')
    break
else
    draw_otbot(m,q)
    if jaux2 == 1
        GBpplot = GBpmat(:,1:jaux);
        multi_circles(GBpplot,m.l_2/6,'b')
        jaux2 = jaux2 + 1;
        jaux = jaux+1;
    elseif jaux2>= Gpf
        multi_circles(GBpplot,m.l_2/6,'b')
        jaux2 = 1;
    else
        multi_circles(GBpplot,m.l_2/6,'b')
        jaux2 = jaux2 + 1;
    end

    hold on
    if strcmp(DistFlag,'YES') && ...
            norm(Dist_funct(times(i))) ~= 0
        Dist_video = Dist_funct(times(i));
        D_XY = Dist_video(1:2,1);
        D_XY = D_XY./norm(D_XY);
        D_XY = 3*m.l_1*D_XY;
        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
            D_XY(1,1)));
        arrows(states(i,1) - D_XY(1,1), ...
            states(i,2) - D_XY(2,1), norm(D_XY), ...
            270 - PolAng , 'FaceColor','r', ...
            'EdgeColor','none');
    end
    hold off

    % 'gcf' can handle if you zoom in to take a movie
        .
    frame = getframe(gcf);
    writeVideo(writerObj, frame);
end
```

```
        end
        close(writerObj); % Saves the movie.

    elseif strcmp(TargetFlag,'YES') && TEFlag == 0
        for i = 1:n:length(times)
                q.x = states(i,1);
                q.y = states(i,2);
                q.alpha = states(i,3);
                q.varphi_r = states(i,4);
                q.varphi_l = states(i,5);
                q.varphi_p = states(i,6);

                elapsed = cputime-time;
                if elapsed > 1200
                    disp(elapsed);
                    disp('took too long to generate the video')
                    break
                else
                    draw_otbot(m,q)
                    if jaux2 == 1
                        GBpplot = GBpmat(:,1:jaux);
                        multi_circles(GBpplot,m.l_2/6,'b')
                        hold on
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = jaux2 + 1;
                        jaux = jaux+1;
                    elseif jaux2>= Gpf
                        multi_circles(GBpplot,m.l_2/6,'b')
                        hold on
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = 1;
                    else
                        multi_circles(GBpplot,m.l_2/6,'b')
                        hold on
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = jaux2 + 1;
                    end

                    hold on
```

```
                        if strcmp(DistFlag,'YES') && ...
                                norm(Dist_funct(times(i))) ~= 0
                            Dist_video = Dist_funct(times(i));
                            D_XY = Dist_video(1:2,1);
                            D_XY = D_XY./norm(D_XY);
                            D_XY = 3*m.l_1*D_XY;
                            PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                                D_XY(1,1)));
                            arrows(states(i,1) - D_XY(1,1), ...
                                states(i,2) - D_XY(2,1), norm(D_XY), ...
                                270 - PolAng , 'FaceColor','r', ...
                                'EdgeColor','none');
                        end
                        hold off

                        % 'gcf' can handle if you zoom in to take a movie
                            .
                        frame = getframe(gcf);
                        writeVideo(writerObj, frame);
                    end
                end
                close(writerObj); % Saves the movie.
            end

        case 'PF'
            time=cputime;
            % Animation
            % h is the sampling time
            % n is the scaling factor in order not to plot with the same step
            % than during the integration with ode45

            fs=30;
            n=round(1/(fs*h));

            % Set up the movie.
            writerObj = VideoWriter('otbot_sim','MPEG-4'); % Name it.
            %writerObj.FileFormat = 'mp4';
            writerObj.FrameRate = fs; % How many frames per second.
            open(writerObj);
            jaux = 1;
            jaux2 = 1;

            TEFlagcheck = strcmp(TargetFlag,'YES') && TEFlag == 1;
            if strcmp(TargetFlag,'NO') || TEFlagcheck
                for i = 1:n:length(times)
```

```
q.x = states(i,1);
q.y = states(i,2);
q.alpha = states(i,3);
q.varphi_r = states(i,4);
q.varphi_l = states(i,5);
q.varphi_p = states(i,6);

elapsed = cputime-time;
if elapsed > 1200
    disp(elapsed);
    disp('took too long to generate the video')
    break
else
    draw_otbot(m,q)
    if jaux2 == 1

        GPpplot = GPpmat(:,1:jaux);

        multi_circles(GPpplot,m.l_2/6,'g')
        jaux2 = jaux2 + 1;
        jaux = jaux+1;
    elseif jaux2>= Gpf

        multi_circles(GPpplot,m.l_2/6,'g')
        jaux2 = 1;
    else

        multi_circles(GPpplot,m.l_2/6,'g')
        jaux2 = jaux2 + 1;
    end

    hold on
    if strcmp(DistFlag,'YES') && ...
            norm(Dist_funct(times(i))) ~= 0
        Dist_video = Dist_funct(times(i));
        D_XY = Dist_video(1:2,1);
        D_XY = D_XY./norm(D_XY);
        D_XY = 3*m.l_1*D_XY;
        PolAng = rad2deg(atan2(-D_XY(2,1), - ...
            D_XY(1,1)));
        arrows(states(i,1) - D_XY(1,1), ...
            states(i,2) - D_XY(2,1), norm(D_XY), ...
            270 - PolAng , 'FaceColor','r', ...
            'EdgeColor','none');
    end
```

```
                    hold off

                    % 'gcf' can handle if you zoom in to take a movie
                        .
                    frame = getframe(gcf);
                    writeVideo(writerObj, frame);
                end
            end
            close(writerObj); % Saves the movie.

        elseif strcmp(TargetFlag,'YES') && TEFlag == 0
            for i = 1:n:length(times)
                q.x = states(i,1);
                q.y = states(i,2);
                q.alpha = states(i,3);
                q.varphi_r = states(i,4);
                q.varphi_l = states(i,5);
                q.varphi_p = states(i,6);

                elapsed = cputime-time;
                if elapsed > 1200
                    disp(elapsed);
                    disp('took too long to generate the video')
                    break
                else
                    draw_otbot(m,q)
                    if jaux2 == 1
                        GPpplot = GPpmat(:,1:jaux);
                        multi_circles(GPpplot,m.l_2/6,'g')
                        hold on
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = jaux2 + 1;
                        jaux = jaux+1;
                    elseif jaux2>= Gpf
                        multi_circles(GPpplot,m.l_2/6,'g')
                        hold on
                        otbot_circle_v2(targetpos(i,1), ...
                            targetpos(i,2),m.l_2/4,'r');
                        hold off
                        jaux2 = 1;
                    else
                        multi_circles(GPpplot,m.l_2/6,'g')
                        hold on
```

```
                                otbot_circle_v2(targetpos(i,1), ...
                                    targetpos(i,2),m.l_2/4,'r');
                                hold off
                                jaux2 = jaux2 + 1;
                            end

                            hold on
                            if strcmp(DistFlag,'YES') && ...
                                    norm(Dist_funct(times(i))) ~= 0
                                Dist_video = Dist_funct(times(i));
                                D_XY = Dist_video(1:2,1);
                                D_XY = D_XY./norm(D_XY);
                                D_XY = 3*m.l_1*D_XY;
                                PolAng = rad2deg(atan2(-D_XY(2,1), - ...
                                    D_XY(1,1)));
                                arrows(states(i,1) - D_XY(1,1), ...
                                    states(i,2) - D_XY(2,1), norm(D_XY), ...
                                    270 - PolAng , 'FaceColor','r', ...
                                    'EdgeColor','none');
                            end
                            hold off

                            % 'gcf' can handle if you zoom in to take a movie
                                .
                            frame = getframe(gcf);
                            writeVideo(writerObj, frame);
                        end
                    end
                    close(writerObj); % Saves the movie.
                end

            otherwise
                disp('This TrackFlag does not exist, omitting the video')
        end
    case 'NO'
        disp('Simulation without video launched');
    otherwise
        disp('This FlagVideo input does not exist');
end
```

A.4.2 Build control

```
function [K] = build_cntrl(eig_set)
%BUILD_CNTRL Summary of this function goes here
%   Detailed explanation goes here
```

```
% Our linear system matrices are

Actc = [zeros(3,3),eye(3,3); zeros(3,3), zeros(3,3)];
Bctc = [zeros(3,3);eye(3,3)];

K = place(Actc,Bctc,eig_set);
end
```

A.4.3 Dynamic equations with control

```
function [ xdot ] = xdot_otbot(t, xs, m, sm, u, u_Flag, cp, ...
    goal_Flag, FrictFlag, DistFlag)
%   Differential equation (non-linearized), that modelizes our Otbot
%   system

% x = xs(1);
% y = xs(2);
alpha = xs(3);
% varphi_r = xs(4);
% varphi_l = xs(5);
varphi_p = xs(6);

% x_dot = xs(7);
% y_dot = xs(8);
alpha_dot = xs(9);
varphi_dot_r = xs(10);
varphi_dot_l = xs(11);
varphi_dot_p = xs(12);

% Setting fricctions
switch FrictFlag
    case 'YES'
        tau_friction(1:3,1) = zeros(3,1);
        tau_friction(4,1) = -m.b_frict(1,1)*varphi_dot_r; % tau_friction_right
        tau_friction(5,1) = -m.b_frict(2,1)*varphi_dot_l; % tau_friction_left
        tau_friction(6,1) = -m.b_frict(3,1)*varphi_dot_p; % tau_friction_pivot
    case 'NO'
        tau_friction = zeros(6,1);
    otherwise
        disp('Warning: This FrictFlag does not exist. System will be' ...
            'simulated without friction')
        tau_friction = zeros(6,1);
end

% Setting Distrurbances
```

```
switch DistFlag
    case 'YES'
        D_vec = Dist_funct(t);
    case 'NO'
        D_vec = zeros(6,1);
    otherwise
        disp('Warning: This DistFlag does not exist. System will be' ...
            'simulated without disturbances')
        D_vec = zeros(6,1);
end

MIIKs = sm.MIIKmatrix(alpha,varphi_p);
M_bars2 = sm.M_barmatrix2(alpha,varphi_p);
C_bars2 = sm.C_barmatrix2(alpha,alpha_dot,varphi_p,varphi_dot_p);
MIIKdots = sm.MIIKdotmatrix(alpha,alpha_dot,varphi_p,varphi_dot_p);
Dmats = sm.Deltamatrix(alpha,varphi_p);

Msys2 = [M_bars2, zeros(3);
         -MIIKs, eye(3)];

qdot = xs(7:12);

switch u_Flag
    case 'CTE'
        qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction + ...
            Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

    case 'VAR'
        u_f= u_function(t,u);
        qdotdot = pinv(Msys2)*[u_f + Dmats.'*tau_friction + ...
            Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

    case 'CTC_PP'
        switch goal_Flag
            case 'FIX'
                pdiff(1:2,1) = cp.pss(1:2,1)-xs(1:2,1);
                pdiff(3,1) = cp.pss(3,1)-xs(3,1);
                pdiff(4:6,1) = cp.pss(4:6,1)-xs(7:9,1);

                u_2 = cp.K*pdiff;
                u = M_bars2*u_2 + C_bars2*qdot(1:3,1);
                qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction + ...
                    Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots* ...
                    qdot(1:3,1)];
```

```
case 'CIRCLE'
    [xss,yss,alphass,xdotss,ydotss,alphadotss,xdotdotss, ...
        ydotdotss,alphadotdotss] = TD_circle_of_time_XYA(t);
    cp.pss = [xss;yss;alphass;xdotss;ydotss;alphadotss];
    pd_dotdot = [xdotdotss; ydotdotss; alphadotdotss];

    pdiff(1:2,1) = cp.pss(1:2,1)-xs(1:2,1);
    pdiff(3,1) = cp.pss(3,1)-xs(3,1);
    pdiff(4:6,1) = cp.pss(4:6,1)-xs(7:9,1);

    u_2 = pd_dotdot + cp.K*pdiff;
    u = M_bars2*u_2 + C_bars2*qdot(1:3,1);
    qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction + ...
        Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots* ...
        qdot(1:3,1)];

case 'POLILINE'
    [xss,yss,alphass,xdotss,ydotss,alphadotss,xdotdotss, ...
        ydotdotss,alphadotdotss] = TD_polyline_of_time(t);
    cp.pss = [xss;yss;alphass;xdotss;ydotss;alphadotss];
    pd_dotdot = [xdotdotss; ydotdotss; alphadotdotss];

    pdiff(1:2,1) = cp.pss(1:2,1)-xs(1:2,1);
    pdiff(3,1) = cp.pss(3,1)-xs(3,1);
    pdiff(4:6,1) = cp.pss(4:6,1)-xs(7:9,1);

    u_2 = pd_dotdot + cp.K*pdiff;
    u = M_bars2*u_2 + C_bars2*qdot(1:3,1);
    qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction + ...
        Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots* ...
        qdot(1:3,1)];

otherwise
    disp('This goal_Flag does not exist (xdot_otbot)')
    disp('Simulating with a FIX point X=2 Y=2 Alpha = 0 with' ...
        'standart PD CTC')
    cp.pss = [2,2,0,0,0,0]';

    pdiff(1:2,1) = cp.pss(1:2,1)-xs(1:2,1);
    pdiff(3,1) = cp.pss(3,1)-xs(3,1);
    pdiff(4:6,1) = cp.pss(4:6,1)-xs(7:9,1);

    u_2 = cp.K*pdiff;
    u = M_bars2*u_2 + C_bars2*qdot(1:3,1);
    qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction + ...
```

```
                    Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

        end

    otherwise
        disp('WARNING THIS FLAG DOES NOT EXIST (xdot_otbot)')
        disp('Simulation with CTE torques will be launched')
        qdotdot = pinv(Msys2)*[u + Dmats.'*tau_friction +  ...
            Dmats.'*D_vec - C_bars2*qdot(1:3,1); MIIKdots*qdot(1:3,1)];

end
xdot=[qdot; qdotdot];
end
```

A.4.4 Circular trajectories

```
function [X,Y,Alpha,X_dot,Y_dot,Alpha_dot,X_dotdot, Y_dotdot, Alpha_dotdot] ...
    = TD_circle_of_time_XYA(t)
%   This function is designed to compute circular trajectories or similar
%   for the Otbot simulations


%---------- Experimental trajectory 1 (Circle) ----------%

% X = sin(0.3*t);
% Y = cos(0.3*t) - 1;
%
% X_dot = 0.3*cos(0.3*t);
% Y_dot = -0.3*sin(0.3*t);
%
% X_dotdot = -0.3^2*sin(0.3*t);
% Y_dotdot = -0.3^2*cos(0.3*t);
%
% Alpha = 0;
% Alpha_dot = 0;
% Alpha_dotdot = 0;

%---------- Experimental trajectory 3 (Infinity Shape) ----------%

t1 = 4;
t2 = (3*pi)/2 + 4;
t3 = 10;
t4 = (9*pi^2-36*pi)/(8-6*pi)+10;
t5 = (12*pi-48)/(4-3*pi) + t4;
```

```
if t <= t1
    Xo = 0;
    Yo = 0;
    Vox = 0;
    Voy = 0;
    ax = 0.25;
    ay = 0;

    X = Xo + Vox*t + 0.5*ax*t^2;
    Y = Yo + Voy*t + 0.5*ay*t^2;

    X_dot = Vox + ax*t;
    Y_dot = Voy + ay*t;

    X_dotdot = ax;
    Y_dotdot = ay;

    Alpha = 0;
    Alpha_dot = 0;
    Alpha_dotdot = 0;

elseif t>t1 && t<=t2
    % Xo = 2;
    % Yo = 0;
    % Vox = 1;
    % Voy = 0;
    thetao = -pi/2;
    omega = 1; % [rad/s]
    xc1 = 2;
    yc1 = 1;
    R = 1;

    X = R*cos(omega*(t-t1) + thetao) + xc1;
    Y = R*sin(omega*(t-t1) + thetao) + yc1;

    X_dot =  - R*omega*sin(thetao + omega*(t - t1));
    Y_dot =    R*omega*cos(thetao + omega*(t - t1));

    X_dotdot =   - R*omega^2*cos(thetao + omega*(t - t1));
    Y_dotdot =   - R*omega^2*sin(thetao + omega*(t - t1));

    Alpha =  omega*(t-t1);
    Alpha_dot =  omega;
    Alpha_dotdot = 0;
```

```
elseif t>t2 && t<=t3
    Xo = 1;
    Yo = 1;
    Vox = 0;
    Voy = -1;
    ax = 0;
    ay = 4*(8-3*pi)/(12-3*pi)^2;

    X = Xo + Vox*(t-t2) + 0.5*ax*(t-t2)^2;
    Y = Yo + Voy*(t-t2) + 0.5*ay*(t-t2)^2;

    X_dot = Vox + ax*(t-t2);
    Y_dot = Voy + ay*(t-t2);

    X_dotdot = ax;
    Y_dotdot = ay;

    Alpha = 3*pi/2;
    Alpha_dot = 0;
    Alpha_dotdot = 0;

elseif t>t3 && t<=t4
    % Xo = 1;
    % Yo = -1;
    % Vox = 0;
    % Voy = (-56+9*pi)/(16-3*pi);
    thetao = 0;
    omega = (4-3*pi)/(12-3*pi); % [rad/s]
    xc2 = 0;
    yc2 = -1;
    R = 1;

    X = R*cos(omega*(t-t3) + thetao) + xc2;
    Y = R*sin(omega*(t-t3) + thetao) + yc2;

    X_dot =  - R*omega*sin(thetao + omega*(t - t3));
    Y_dot =    R*omega*cos(thetao + omega*(t - t3));

    X_dotdot =   - R*omega^2*cos(thetao + omega*(t - t3));
    Y_dotdot =   - R*omega^2*sin(thetao + omega*(t - t3));

    Alpha =  omega*(t-t3) + 3*pi/2;
    Alpha_dot =  omega;
    Alpha_dotdot = 0;
```

```
elseif t>t4 && t<=t5
    Xo = 0;
    Yo = 0;
    Vox = -((4-3*pi)/(12-3*pi));
    Voy = 0;
    Xf = 2;
    ax = 2*(Xf - Xo - Vox*(t5-t4))/(t5-t4)^2;
    ay = 0;

    X = Xo + Vox*(t-t4) + 0.5*ax*(t-t4)^2;
    Y = Yo + Voy*(t-t4) + 0.5*ay*(t-t4)^2;

    X_dot = Vox + ax*(t-t4);
    Y_dot = Voy + ay*(t-t4);

    X_dotdot = ax;
    Y_dotdot = ay;

    Alpha = 0;
    Alpha_dot = 0;
    Alpha_dotdot = 0;
else
    X = 2;
    Y = 0;

    X_dot = 0;
    Y_dot = 0;

    X_dotdot = 0;
    Y_dotdot = 0;

    Alpha = 0;
    Alpha_dot = 0;
    Alpha_dotdot = 0;
end

%-------------------------------------------------------------------------%


end
```

A.4.5 Polyline trajectories

```
function [X,Y,Alpha,X_dot,Y_dot,Alpha_dot, X_dotdot, Y_dotdot, Alpha_dotdot] ...
    = TD_polyline_of_time(t)
```

```
%   This function is designed to compute trajectories for the Otbot
%   simulations

% Polyline type 1 verical M letter
% if t <= 5
%     X =(1/5)*t;
%     Y = 0;
%     X_dot = (1/5);
%     Y_dot = 0;
% elseif t>5 && t<=10
%     X = -(1/5)*(t-5) + 1;
%     Y = (1/5)*(t-5);
%     X_dot = -(1/5);
%     Y_dot = (1/5);
% elseif t>10 && t<=15
%     X = (1/5)*(t-10);
%     Y = (1/5)*(t-5);
%     X_dot = (1/5);
%     Y_dot = (1/5);
% elseif t>15 && t<=20
%     X = -(1/5)*(t-15) + 1;
%     Y = 2;
%     X_dot = -(1/5);
%     Y_dot = 0;
% else
%     X = 0;
%     Y = 2;
%     X_dot = 0;
%     Y_dot = 0;
% end

% Polyline Type 2 Square signal
% if t <= 5
%     X = (1/5)*t;
%     Y = 0;
%     X_dot = 1/5;
%     Y_dot = 0;
% elseif t>5 && t<=10
%     X = 1;
%     Y = (1/5)*(t-5);
%     X_dot = 0;
%     Y_dot = 1/5;
% elseif t>10 && t<=15
%     X = (1/5)*(t-10)+1;
%     Y = 1;
```

```
%     X_dot = 1/5;
%     Y_dot = 0;
% elseif t>15 && t<=20
%     X = 2;
%     Y = -(1/5)*(t-15)+1;
%     X_dot = 0;
%     Y_dot = -1/5;
% elseif t>20 && t<=25
%     X = (1/5)*(t-20)+2;
%     Y = 0;
%     X_dot = 1/5;
%     Y_dot = 0;
% else
%     X = 3;
%     Y = 0;
%     X_dot = 0;
%     Y_dot = 0;
% end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Type 3 Folded M
% if t <= 5
%     X =(1/5)*t;
%     Y = 0;
%     X_dot = (1/5);
%     Y_dot = 0;
% elseif t>5 && t<=10
%     X = -(1/5)*(t-5) + 1;
%     Y = 0;
%     X_dot = -(1/5);
%     Y_dot = 0;
% elseif t>10 && t<=15
%     X = (1/5)*(t-10);
%     Y = 0;
%     X_dot = (1/5);
%     Y_dot = 0;
% elseif t>15 && t<=20
%     X = -(1/5)*(t-15) + 1;
%     Y = 0;
%     X_dot = -(1/5);
%     Y_dot = 0;
% else
%     X = 0;
%     Y = 0;
%     X_dot = 0;
```

```
%     Y_dot = 0;
% end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% X_dotdot = 0;
% Y_dotdot = 0;
%
% Alpha = 0;
% Alpha_dot = 0;
% Alpha_dotdot = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constant acceleration

% Set acceleration value
% a = 0.5; % [m/s^2]
%
% X = 0.5*a*t^2;
% Y = 0;
%
% X_dot = a*t;
% Y_dot = 0;
%
% X_dotdot = 0;
% Y_dotdot = 0;
%
% Alpha = 0;
% Alpha_dot = 0;
% Alpha_dotdot = 0;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Variable acceleration (Acceleration function of time)

% Set acceleration value
a = 0.5; % [m/s^2]

X = (1/6)*a*t^3;
Y = 0;

X_dot = 0.5*a*t^2;
Y_dot = 0;

X_dotdot = 0;
Y_dotdot = 0;
```

```
Alpha = 0;
Alpha_dot = 0;
Alpha_dotdot = 0;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

end
```

B
Project budget

The budget presented below evaluates the cost of developing the software developed, both in human and material resources.

B.1 Initial estimations

In this section we will specify the hourly costs for the preparation of the budget. We consider two types of costs:

- The hourly cost of staff.
- The hourly cost of using the equipment.

B.1.1 The hourly cost of staff

We estimate the following salaries:

$$\begin{array}{ll} Analyst & \ldots \quad 15.62\ EUR/hour \\ Programmer & \ldots \quad 13.97\ EUR/hour \\ Operator & \ldots \quad 10.41\ EUR/hour \end{array}$$

B.1.2 Hourly cost of using the equipment

For the assessment of the hourly cost of using the equipment it is necessary to take into account: its amortization, the cost of the personnel of the Institute of Robotics and Industrial Informatics (CSIC-UPC) necessary for the good operation of the equipment, and maintenance contracts, if any.

The only equipment used in this project is a laptop PC with the following associated costs:

1. **Equipment cost:** The PC used includes an HP ProDesk 400 G4 PC tower, with 7,200 rpm SATA hard drive, 1TB 3.5", Intel HD Graphics 630 board, Realtek RTL8111 HSH GbE LOM network card, 2 USB 3.1 Gen1 ports, 4 USB ports 2.0, 3.4GHz-3.8Ghz Intel Core i5-7500 CPU, 8Gb DDR4-2400 (1 x 8Gb) SDRAM, HP USB Business Compact Keyboard, HP USB Optical Mouse, 19.5"HP 20kd Monitor (1440x900 at 60Hz), and Windows 10 operating system. The cost of this computer is approximately 746,75 EUR. This amount also includes the installation costs of the machine. If we consider that we want to repay

the equipment in five years, with an interest rate of 15%, the amortization year results from:

$$C(equipment) = 746.75 \times \frac{0.15 \times 1.15^5}{1.15^5 - 1} = 222.76\ EUR/year$$

2. **Maintenance:** The annual maintenance cost of the complete equipment is estimated at 10% of its purchase price:

$$C(maintenance) = 746.75 \times 0.1 = 74.67\ EUR/year$$

3. **Power consumption:** It is evaluated in:

$$C(consumption) = 100\ EUR/year$$

4. **Cost of IRI staff:** The costs of an operator working 0.5 hours per week for 40 weeks each year are considered:

$$C(pers\ IRI) = 208\ EUR/year$$

Based on the above partial costs, the total annual cost will be:

$C(equipment)$	...	$222.76\ EUR/year$
$C(maintenance)$	...	$74.67\ EUR/year$
$C(pers\ IRI)$	...	$208\ EUR/year$
$C(consumption)$	...	$100\ EUR/year$
$C(annual)$	...	$605.43\ EUR/year$

The hourly cost of CPU, considering that they work 8 hours/day and 5 days/week for 40 weeks/year and the time of use of the CPU is 60%, it is:

$$C(hourlyCPU) = \frac{605.43}{8 \times 5 \times 40 \times 0.6} = 0.63\ EUR/hour$$

B.2 Development cost

Taking into account the prices stipulated in the previous section, we will then calculate the cost of developing the project, ie the cost of staff, equipment and miscellaneous expenses. It should be noted that the costs related to the fixed assets of the IRI are not taken into account, but only those for the development of the project in terms of human resources and computer equipment.

B.2.1 Staff cost

It is estimated that the time invested in the project has been 23 weeks with full dedication of 40 hours/week. The project involved a person earning the salary of a computer analyst:

$$C(staff) = 920\ hours \times 15.62\ EUR/hour = 14,370.4\ EUR$$

B.2.2 Cost of using the equipment

It is estimated that 60% of the total project time has been spent implementing the programs. In fact, you have to deduct 35% of that time for CPU downtime on your computer. It is considered that when using the CPU, we actually use 80% of it (despite being multi-user, the usage regime is practically single-user). Therefore, we can evaluate the following times:

$$\begin{array}{lrcr} Computer\ time & 920 \times 0.60 & = & 552\ hours, \\ Useful\ computer\ time & 552 \times 0.65 & = & 358.80\ hours, \\ CPU\ time & 358.80 \times 0.80 & = & 287.04\ hours, \end{array}$$

and the CPU cost turns out to be

$$C(CPU) = 287.04 \times 0.63 = 180.83\ EUR.$$

B.2.3 Documentation and printing costs

The following concepts are considered:

$$\begin{array}{lcr} Documentation & \ldots & 90.00\ EUR \\ Paper\ and\ printing & \ldots & 15.00\ EUR \\ \hline C(various) & & 105.00\ EUR \end{array}$$

B.2.4 Total cost of development

The total cost of developing the project, finally, is the sum of the above costs:

$$\begin{array}{lcr} C(staff) & \ldots & 14,370.40\ EUR \\ C(CPU) & \ldots & 180.83\ EUR \\ C(various) & \ldots & 105.00\ EUR \\ \hline C(TOTAL) & \ldots & 14,656.23\ EUR \end{array}$$