Ecole Polytechnique Fédérale de Lausanne

Projet de Master RO, 2021, Section de Microtechnique

Planning and Control of Optimal Trajectories for an Omnidirectional Tire-wheeled Robot

EPFL Professor:
Giancarlo Ferrari Trecate

Author:
Maxime Gautier

External Experts:
Lluís Ros
Enric Celaya

March 12, 2021

Work done at
Universitat Politècnica de Catalunya
in
Institut de Robòtica i Informàtica Industrial

Contents

Chapter 1

Introduction

A large variety of mobile robot designs exists today [1, 2], often tailored to specific applications. Of those vehicles, only a few are omnidirectional, despite the clear benefits in motion, adaptability and trajectory following [3]. Indeed, omnidirectional robots are able to follow any complex trajectory and can easily perform tasks in environments with static or moving obstacles. Such environments, like warehouses or factory workshops are actually in the process of automatizing their workflow by implementing collaborative robots to deliver, transport and store inventory. Some key desired traits of such robots include high mobility and autonomous adaptive behavior to their environment.

Omnidirectionality in a mobile robot is therefore often a desirable characteristic in a range of applications. A typical method to achieve omnidrectional behavior in a robot is to use special wheels, like Mecanum or Omni wheels [4, 5, 6]. These are complex mechanisms which are designed to allow their equipped vehicle movement in any direction on the 2D plane. They work quite well, however they present some inherent disadvantages such as difficulties in their manufacturing, complexity of control and low load-bearing capabilities. Fortunately, there exist other methods to achieve omnidirectionality in robotic vehicles and avoid those issues, such as structuring a mobile robot to have an omnidirectional component, rather than using wheels to make the entire robot omnidirectional. The specific design treated in this thesis is one of a tire-wheeled differential drive robot topped with an omnidirectional platform.

The design of this omnidirectional tire-wheeled robot dates back to a couple of decades ago with an intent of application in robot soccer [7, 8, 9]. Along with the structural design, the authors developed a kinematic model and a fuzzy logic based path planner, which is enough to show the potential of this robot. However, since few direct applications were possible at the time, little work concerning further modeling, kinodynamic planning and control of this robot has been done ever since.

The purpose of this project is therefore to plan and follow optimal trajectories for an omnidirectional tire-wheeled robot. To do this, we must first construct a more generic kinematic model, followed by a generic dynamic model for this robotic structure. This generality will allow easy changes in the dimensions and weight distribution when constructing and using this robot, as well as the synthesis of robot trajectories minimizing time, control effort, or even the energy required by the robot if necessary.

Once the complete dynamical model is built, we can use it to plan optimal trajectories. For this purpose, we will use trajectory optimization methods [10, 11] and more specifically direct collocation techniques [12] since these are easier to pose and solve and allow for a great versatility in constraints implementation. This flexibility allows us to model any desired environment in which to plan a trajectory, as well as to impose any restrictions we wish on the modelled robot. In fact, the wide array of possible constraints which can be expressed using this paradigm is the reason it was chosen for this project.

Finally, in order to follow the previously-planned optimal trajectories, we designed a controller for the robot. To ensure any trajectory can be matched from any position, the implemented controller must be globally stable. With that in mind, we designed a computed-torque controller capable of tracking any desired trajectories in task space, and compensate unforeseen disturbances.

Chapter 2

Kinematic Model

A kinematic model describes the motion of a robot mechanism regardless of forces and torques that cause it. This section will start with a description of the geometry of the robot studied during this project, as well as the state coordinates used to model it. From this geometry, we can deduce kinematic constraints which will allow us to find solutions for the instantaneous kinematic problems, both forward and inverse.

2.1 Introduction to the robot

2.1.1 Reference frames and vector bases

The robot studied in this project is an omnidirectional tire-wheeled robot, called Otbot for short. It is composed of a chassis topped by a rotating platform. The chassis (Fig. 2.1 **(a)**) is a classic differential drive robot, with two wheels powered by DC motors and passive caster wheels for horizontal stability (not drawn). The circular platform (Fig. 2.1 **(b)**) mounted above the chassis is rotated through a pivot joint in its center controlled through another DC motor (Fig. 2.1 **(c)**). The pivot joint is offset from the wheel axis, which allows the platform to behave like an omnidirectional vehicle in the 2D horizontal plane.

To obtain the kinematic and dynamic models of the robot, we will use the reference frames drawn in Fig. 2.2. The blue frame is the absolute frame fixed to the ground. The

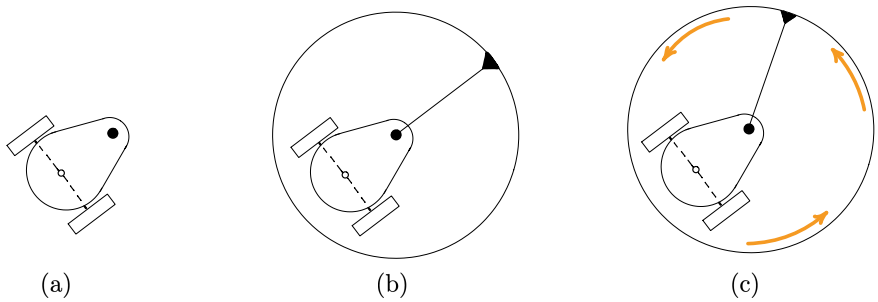


Figure 2.1 – Kinematic structure of Otbot.

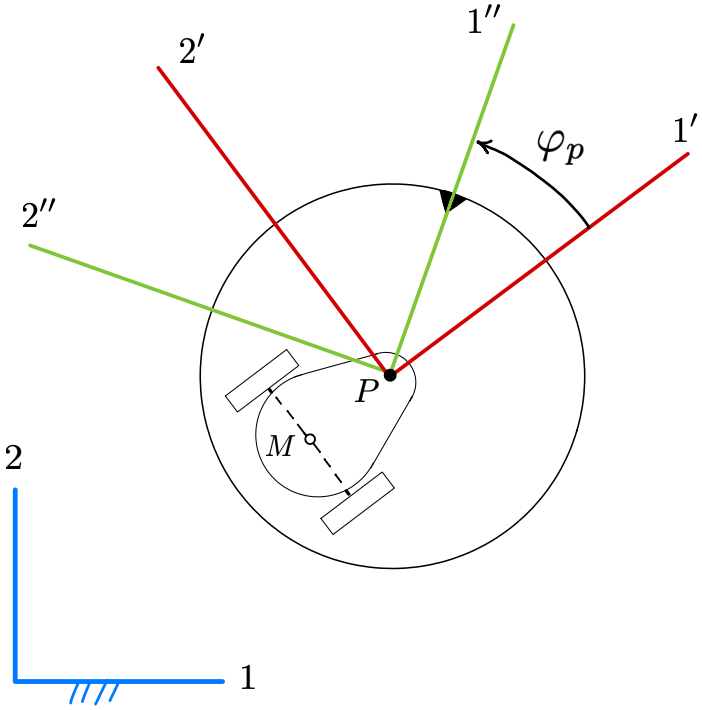


Figure 2.2 – Reference frames, vector bases and main nomenclature used in the elaboration of mathematical models.

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.

2.1.2 Configuration and state coordinates

The robot configuration, as shown in Fig. 2.3, can be described by the following six coordinates:

- 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

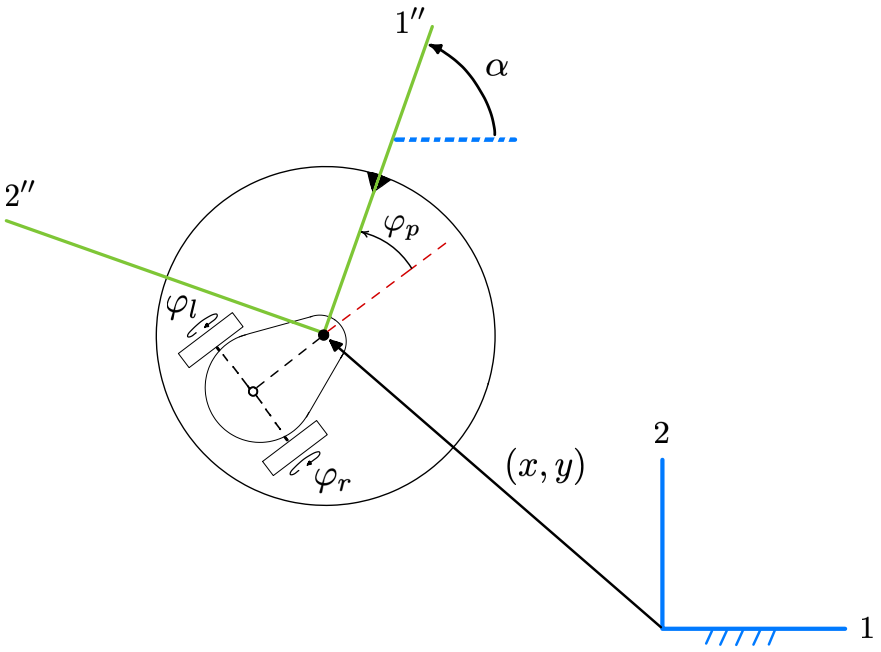


Figure 2.3 – Otbot's configuration and state coordinates.

The robot configuration is thus given by

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

with x, y, α, the 'platform' coordinates and $\varphi_r, \varphi_l, \varphi_p$ the 'motor' coordinates. The time derivative of $\mathbf{q}$ provides the robot velocity[1]

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

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 imposed by rolling contacts

The rolling contacts of the wheels impose a kinematic constraint of the form:

$$\mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} = \mathbf{0}, \tag{2.2.1}$$

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.

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.

[1] In this report, column vectors will often be represented using parentheses for a cleaner readability

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. 2.4. 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 offset of the pivot point P 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 the basis B' = {1',2',3'} we have

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

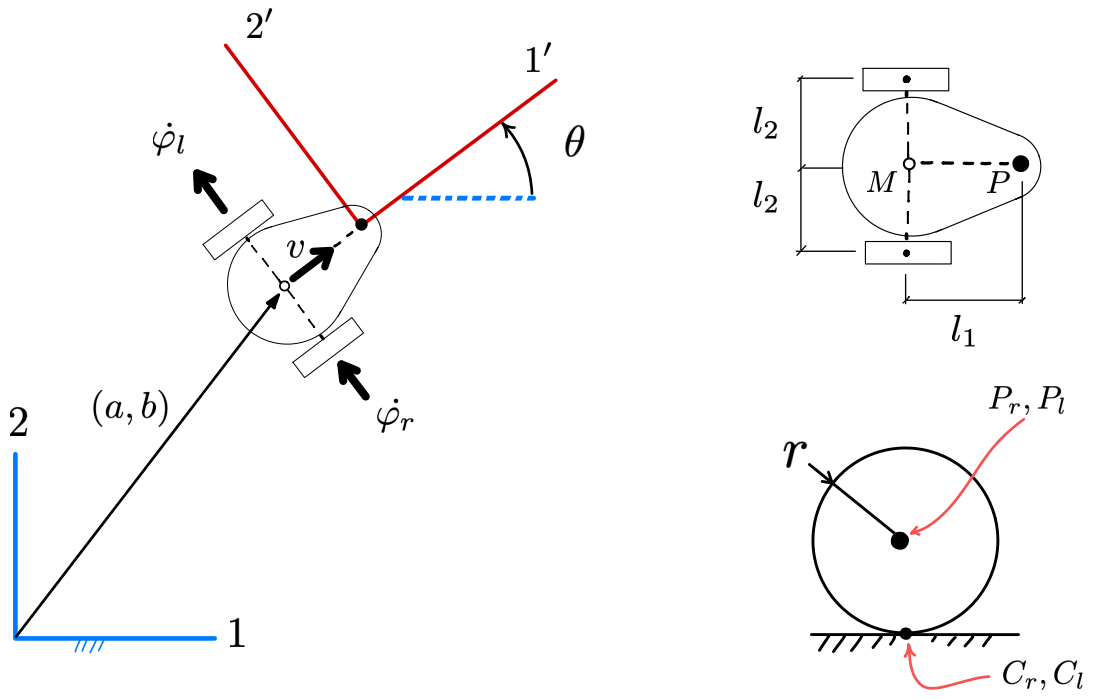


Figure 2.4 – The robot chassis and its rolling contacts with the non-slip condition.

Also note that,

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

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

The velocities of the ground contact points are thus given by

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

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

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:

$$v = \frac{r\,(\dot{\varphi}_l + \dot{\varphi}_r)}{2} \tag{2.2.2}$$

$$\dot{\theta} = -\frac{r\,(\dot{\varphi}_l - \dot{\varphi}_r)}{2l_2}. \tag{2.2.3}$$

Note from Fig. 2.4 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. (2.2.2) in these two equations, and also considering Eq. (2.2.3), we obtain the system:

$$\begin{cases} \dot{a} = \cos(\theta)\left(\dfrac{r\dot{\varphi}_l}{2} + \dfrac{r\dot{\varphi}_r}{2}\right) \\ \dot{b} = \sin(\theta)\left(\dfrac{r\dot{\varphi}_l}{2} + \dfrac{r\dot{\varphi}_r}{2}\right) \\ \dot{\theta} = -\dfrac{r(\dot{\varphi}_l - \dot{\varphi}_r)}{2l_2} \end{cases} \tag{2.2.4}$$

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.

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 of equations 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 $\mathbf{x}$.

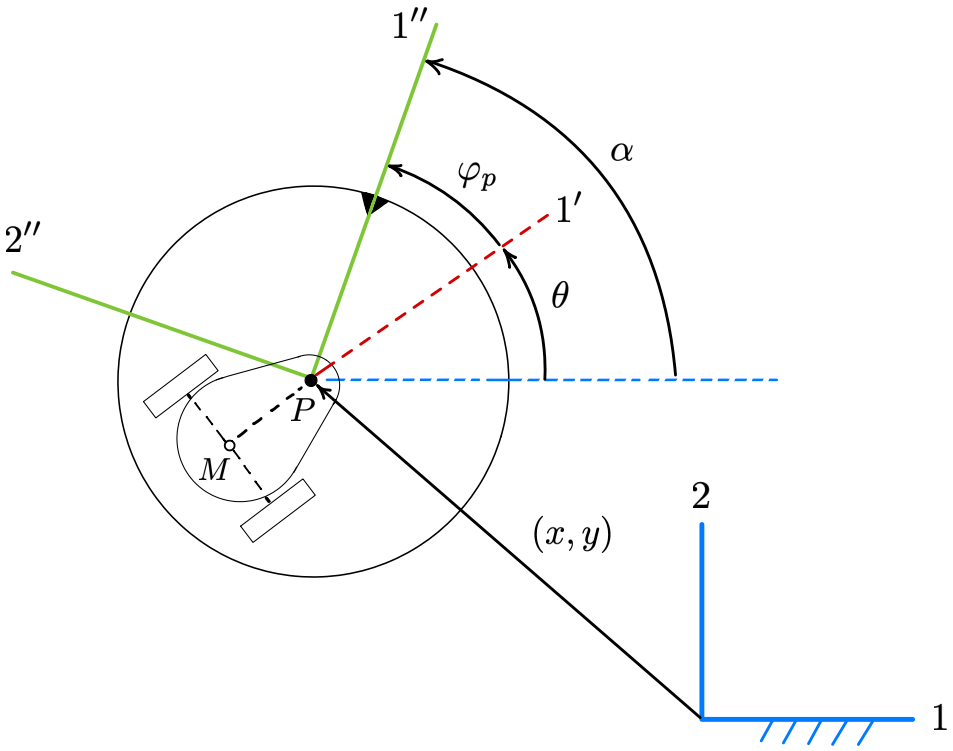


Figure 2.5 – Relationship between α, φ_p, and θ.

As it is shown in Fig. 2.5, 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 Eq. (2.2.4) above to obtain:

$$\dot{x} + l_1 \sin(\alpha - \varphi_p)\ (\dot{\alpha} - \dot{\varphi}_p) = \cos(\alpha - \varphi_p)\ \left(\frac{r\dot{\varphi}_l}{2} + \frac{r\dot{\varphi}_r}{2}\right), \tag{2.2.5}$$

$$\dot{y} - l_1 \cos(\alpha - \varphi_p)\ (\dot{\alpha} - \dot{\varphi}_p) = \sin(\alpha - \varphi_p)\ \left(\frac{r\dot{\varphi}_l}{2} + \frac{r\dot{\varphi}_r}{2}\right), \tag{2.2.6}$$

$$\dot{\alpha} - \dot{\varphi}_p = -\frac{r\,(\dot{\varphi}_l - \dot{\varphi}_r)}{2\,l_2}. \tag{2.2.7}$$

These are the *velocity constraints* of the Otbot expressed using the state coordinates. 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

$$\begin{bmatrix} 1 & 0 & l_1 s_{\alpha-\varphi_p} & -\frac{1}{2} r c_{\alpha-\varphi_p} & -\frac{1}{2} r c_{\alpha-\varphi_p} & -l_1 s_{\alpha-\varphi_p} \\ 0 & 1 & -l_1 c_{\alpha-\varphi_p} & -\frac{1}{2} r s_{\alpha-\varphi_p} & -\frac{1}{2} r s_{\alpha-\varphi_p} & l_1 c_{\alpha-\varphi_p} \\ 0 & 0 & 1 & -\dfrac{r}{2l_2} & \dfrac{r}{2l_2} & -1 \end{bmatrix}$$

in which $s_{\alpha-\varphi_p}$ and $c_{\alpha-\varphi_p}$ are a shorthand for the sine and cosine of the shown angle.

At this point, it is worth noting that Eq. (2.2.7) is actually integrable, which results in an holonomic constraint that relates all of the angles of the robot. After taking the time integral of Eq. (2.2.7) and rearranging the terms, this constraint can be written as

$$\alpha - \frac{r}{2\ell_2}\varphi_r + \frac{r}{2\ell_2}\varphi_e - \varphi_p + K_0 = 0, \tag{2.2.8}$$

where $K_0 = \alpha(0) - \frac{r}{2\ell_2}\varphi_r(0) + \frac{r}{2\ell_2}\varphi_e(0) - \varphi_p(0)$. While Eq. (2.2.8) will not be necessary to obtain the dynamical model of the robot in Chapter 3, it will play an important role in Chapter 4 to construct a new model in independent state coordinates. This new model will be crucial to ensure accurate trajectory optimizations respecting all kinematic constraints of the robot.

For ease of explanation, the system formed by Eqs. (2.2.5) to (2.2.8) the *kinematic constraints*, will be written as follows hereafter

$$\boldsymbol{F}(\boldsymbol{x}) = 0 \tag{2.2.9}$$

2.3 Solution to instantaneous kinematic problems

The robot configuration variables 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 variables of the robot. Their time derivatives $\dot{\boldsymbol{\varphi}}$ and $\dot{\mathbf{p}}$ are called the motor speeds and the platform velocity (or twist) respectively, and their relationship is key to understanding and modelling the behavior of the Otbot.

In this section, we therefore tackle the following two problems whose solution will later be needed to develop the dynamical model of the robot:

- 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 $\dot{\mathbf{p}}$.

We will also show that the IIKP is always solvable, irrespective of the robot configuration, which proves that Otbot's platform 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. (2.2.5), (2.2.6) and (2.2.7) defining $\mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} = \mathbf{0}$:

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

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

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

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

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \end{bmatrix} = \mathbf{M}_{FIK}(\mathbf{q}) \cdot \begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix}, \tag{2.3.1}$$

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

$$\begin{bmatrix} \dfrac{l_2\, r\cos(\alpha - \varphi_p) - l_1\, r\sin(\alpha - \varphi_p)}{2\, l_2} & \dfrac{l_2\, r\cos(\alpha - \varphi_p) + l_1\, r\sin(\alpha - \varphi_p)}{2\, l_2} & 0 \\ \dfrac{l_1\, r\cos(\alpha - \varphi_p) + l_2\, r\sin(\alpha - \varphi_p)}{2\, l_2} & -\dfrac{l_1\, r\cos(\alpha - \varphi_p) - l_2\, r\sin(\alpha - \varphi_p)}{2\, l_2} & 0 \\ \dfrac{r}{2\, l_2} & -\dfrac{r}{2\, l_2} & 1 \end{bmatrix}$$

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

$$\begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \\ \dot{\varphi}_p \end{bmatrix} = \mathbf{M}_{IIK}(\mathbf{q}) \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\alpha} \end{bmatrix}, \tag{2.3.2}$$

where $\mathbf{M}_{IIK}(\mathbf{q}) = \mathbf{M}_{FIK}^{-1}(\mathbf{q})$. At this point, it is important to note that the determinant of $\mathbf{M}_{FIK}(\mathbf{q})$ is

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

Thus, 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.

By taking the symbolic inverse of $\mathbf{M}_{FIK}(\mathbf{q})$ we obtain that $\mathbf{M}_{IIK}(\mathbf{q})$ is:

$$\begin{bmatrix} \frac{l_1 \cos(\alpha-\varphi_p) - l_2 \sin(\alpha-\varphi_p)}{l_1 r} & \frac{l_2 \cos(\alpha-\varphi_p) + l_1 \sin(\alpha-\varphi_p)}{l_1 r} & 0 \\ \frac{l_1 \cos(\alpha-\varphi_p) + l_2 \sin(\alpha-\varphi_p)}{l_1 r} & -\frac{l_2 \cos(\alpha-\varphi_p) - l_1 \sin(\alpha-\varphi_p)}{l_1 r} & 0 \\ \frac{\sin(\alpha-\varphi_p)}{l_1} & -\frac{\cos(\alpha-\varphi_p)}{l_1} & 1 \end{bmatrix} \tag{2.3.3}$$

2.4 The kinematic model in control form

The following 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,

$$\dot{\mathbf{q}} = \mathbf{f}_{\text{kin}}(\mathbf{q}, \mathbf{v}), \tag{2.4.1}$$

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

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

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

The model in Eq. (2.4.1) could be used for trajectory planning already. In doing so, we would neglect the system dynamics, but a strong point would be 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. This model would be useful for proof of concept and early testing of this robot design, which has already been done in the context of this thesis. Our interest here is however in obtaining a more complete and accurate model so we can simulate our robot as realistically as possible. Furthermore, since a kinematic model does not take into account any mass or forces, we could not plan for trajectories minimizing the motor forces, or the energy consumed by the robot, nor take into account frictions effects when needed. To pally the limitations of this kinematic model, we will therefore develop a complete dynamical model of the Otbot.

Chapter 3

Dynamic Model

In order to obtain a more realistic simulation of the Otbot, we must compute a model which takes into account all forces present in the system and their evolution through time. This is called the *dynamic model*. To write it down, we must first find the mass matrix of the system by computing its kinetic energy relative to the absolute frame, which can be considered to be inertial. Since the robot is designed to move on flat terrain, the potential energy will stay constant and thus its generalized conservative force will be null in the dynamical model. Therefore, we will next obtain the kinetic energy of he Otbot and the associate mass matrix. Once these are obtained, we will be able to define each variable present in Lagrange's equation of motion.

This equation can then be used to compute the inverse dynamics, which provides the control actions u that are required to obtain a given acceleration $\ddot{\boldsymbol{q}}$ of the system. In the case of the Otbot, such actions are given by the torques τ_r, τ_l, τ_p applied by the right, left and pivot motors respectively. A further manipulation of the resulting equations allows us to find the forward dynamics of the system, which gives the acceleration $\ddot{q}$ produced by a given action $u = [\tau_r, \tau_l, \tau_p]$. The solution of the forward dynamics can be augmented to obtain the dynamic model of the whole system in the usual control form: $\dot{\mathbf{x}} = \boldsymbol{f}(\mathbf{x}, \mathbf{u})$. It is using this model that the trajectory optimization methods of Chapter 4 will be build upon.

3.1 Kinetic energy of the system

The mass matrix $\mathbf{M}(\mathbf{q})$ describes the mass of our entire dynamical system with regards to the state variables. It can be computed by writing the kinetic energy T of the robot in terms of $\mathbf{q}$ and $\dot{\mathbf{q}}$, and expressing it in the form [13]:

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

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.

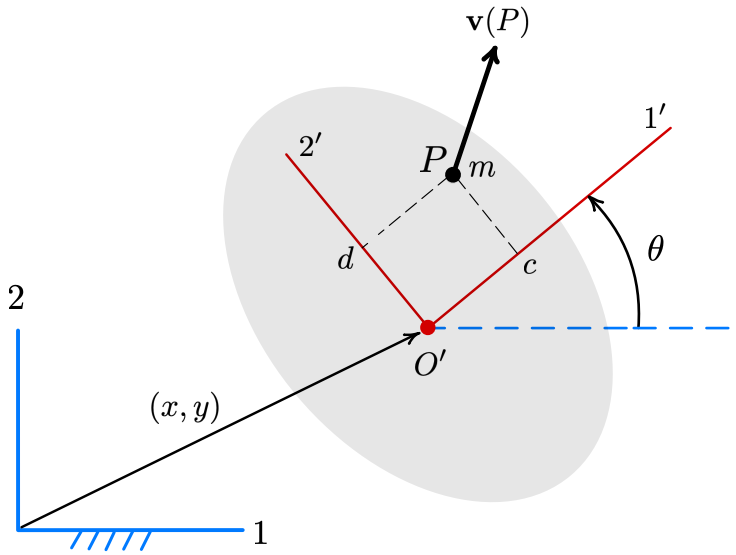


Figure 3.1 – Notation used to derive a generic expression for the translational kinetic energy of a point of mass m located at a point P of a body moving under planar motion (shown in grey). The velocity of the body is given by $\dot{x}$, $\dot{y}$ and $\dot{\theta}$. The coordinates of P in the body-fixed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

We want to build a dynamical model for any arbitrarily placed center of mass of the chassis and platform. We therefore wish to compute the translational kinetic energy T_t of a point mass m located at some point P of a body under planar motion. We wish to express T_t in terms of $(\dot{x}, \dot{y}, \dot{\theta})$, the twist of the body. We can represent this problem using Fig. 3.1 where (c, d) are the coordinates of P in the body reference frame $0', 1', 2'$ and $\vec{\boldsymbol{v}}(P)$ is the absolute velocity of P. To obtain T_t, we first express $\vec{\boldsymbol{v}}(P)$ as a function of (c, d) and $\dot{\theta}$ using the following relation:

$$\vec{\boldsymbol{v}}(P) = \vec{\boldsymbol{v}}(O') + \vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{O'P}}$$

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

By adding $\dot{\theta}$ in the third row, we can obtain the following relationship, which will be useful below:

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

Now note that T_t can be expressed as follows in terms of t:

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

Using the relation $\boldsymbol{t} = \boldsymbol{A} \cdot \dot{\boldsymbol{p}}$, we can now express T_t as:

$$T_t = \frac{1}{2}\dot{\boldsymbol{p}}^{\boldsymbol{T}} \underbrace{\boldsymbol{A}^T \boldsymbol{M'} \boldsymbol{A}}_{\boldsymbol{M_t}} \dot{\boldsymbol{p}} = \frac{1}{2}\dot{\boldsymbol{p}}^{\boldsymbol{T}} \cdot \boldsymbol{M_t} \cdot \dot{\boldsymbol{p}},$$

where we have

$$\begin{aligned} \boldsymbol{M_t}((c,d),m) &= \boldsymbol{A}^T \boldsymbol{M'} \boldsymbol{A} = \\ &= \begin{bmatrix} m & 0 & -cm\sin\theta - dm\cos\theta \\ 0 & m & cm\cos\theta - dm\sin\theta \\ -cm\sin\theta - dm\cos\theta & cm\cos\theta - dm\sin\theta & m(c^2+d^2) \end{bmatrix}. \end{aligned}$$

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

$$T_t(\mathbf{q}, \dot{\mathbf{q}}) = \frac{1}{2}\ \dot{\mathbf{p}}^T \cdot \mathbf{M_t}((c,d),m) \cdot \dot{\mathbf{p}}.$$

We use this formula to compute the expression of translational kinetic energy for each of the four bodies of the Otbot. Their respective expressions can be found in the Appendix A.

The rotational kinetic energy of each body in the Otbot, on the other hand, is obtained with the formula

$$T_r(\mathbf{q}, \dot{\mathbf{q}}) = \frac{1}{2}\ I \cdot \omega^2,$$

where I is the moment of inertia around the axis of rotation and ω the angular velocity of the body. The angular velocities of all bodies are easily obtained from the state variables in $\dot{\mathbf{q}}$. The resulting expressions for the rotational kinetic energy for each of the four bodies can be found in the Appendix A.

Finally we can find the expression for the total kinetic energy with regards to the state coordinates, which is the sum of the translational and rotational energies of each body. This expression of T can be found in the Appendix A. Since T is a quadratic form, the Hessian of T will yield the desired mass matrix $\boldsymbol{M(q)}$ for the system.

Symbol	Meaning
(x_G, y_G)	Coords. of G in the chassis frame F_c
(x_F, y_F)	Coords. of F in the platform frame F_p
m_b	Mass of the chassis base
m_w	Mass of a wheel
m_p	Mass of the platform
I_b	Central moment of inertia of the chassis at G
I_p	Central moment of inertia of the platform at F
I_a	Axial moment of inertia of a wheel
I_t	Twisting moment of inertia of a wheel
l_1	Pivot offset relative to the wheels axis
l_2	One half of the wheels separation
r	Wheel radius

Table 3.1 – Robot parameters symbol and meaning as used in dynamic model expression

Here, we are creating a generic dynamic model which can be used for any desired center of mass of the chassis and platform, which are referenced as G and F respectively. The different robot parameters used in this dynamic model as well as their respective meanings are listed in Table 3.1.

Using the notation defined in Table 3.1 for the robot parameters, we obtain the following expression for $\boldsymbol{M(q)}$:

$$\begin{bmatrix}
m+m_p & 0 & \begin{matrix}2l_1m_ws_\theta\\ -m_p(x_{F}s_\alpha+y_{F}c_\alpha)\\ -m_b(y_{G}c_\theta+x_{G}s_\theta)\end{matrix} & 0 & 0 & \begin{matrix}-2l_1m_ws_\theta\\ +m_b(y_{G}c_\theta+x_{G}s_\theta)\end{matrix}\\
0 & m+m_p & \begin{matrix}-2l_1m_wc_\theta\\ m_p(x_{F}c_\alpha-y_{F}s_\alpha)\\ +m_b(x_{G}c_\theta-y_{G}s_\theta)\end{matrix} & 0 & 0 & \begin{matrix}2l_1m_wc_\theta\\ +m_b(y_{G}s_\theta-x_{G}c_\theta)\end{matrix}\\
\begin{matrix}2l_1m_ws_\theta\\ -m_p(x_{F}s_\alpha+y_{F}c_\alpha)\\ -m_b(y_{G}c_\theta+x_{G}s_\theta)\end{matrix} & \begin{matrix}-2l_1m_wc_\theta\\ m_p(x_{F}c_\alpha-y_{F}s_\alpha)\\ +m_b(x_{G}c_\theta-y_{G}s_\theta)\end{matrix} & 2I_t'+I_p'+I_b' & 0 & 0 & -2I_t'-I_b'\\
0 & 0 & 0 & I_a & 0 & 0\\
0 & 0 & 0 & 0 & I_a & 0\\
\begin{matrix}-2l_1m_ws_\theta\\ +m_b(y_{G}c_\theta+x_{G}s_\theta)\end{matrix} & \begin{matrix}2l_1m_wc_\theta\\ +m_b(y_{G}s_\theta-x_{G}c_\theta)\end{matrix} & -2I_t'-I_b' & 0 & 0 & 2I_t'+I_b'
\end{bmatrix}$$

where

$$
\begin{aligned}
s_\theta &= \sin\theta, \\
c_\theta &= \cos\theta, \\
\theta &= \alpha - \varphi_p, \\
m &= m_b + 2m_w, \\
I_t' &= I_t + m_w(l_1^2 + l_2^2), \\
I_p' &= I_p + m_p(x_F^2 + x_F^2), \\
I_b' &= I_b + m_b(x_G^2 + x_G^2).
\end{aligned}
$$

3.2 Lagrange's equation of motion

Lagrange's equation of motion for a robot like the Otbot, which is subject to holonomic and nonholonomic constraints, takes the form

$$\mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\,\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}(\mathbf{q})^T\,\boldsymbol{\lambda} = \mathbf{E}(\mathbf{q})\mathbf{u} + \mathbf{Q}_f, \tag{3.2.1}$$

where:

- $\mathbf{q}$ is the configuration vector of the robot (of size $n_q = 6$ in our case)
- $\mathbf{u} = [\tau_r, \tau_l, \tau_p]^T$ 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 6×6).
- $\mathbf{C}\,(\mathbf{q},\dot{\mathbf{q}})$ is the generalized Coriolis 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 multipliers
- $\mathbf{E}(\mathbf{q})\cdot\mathbf{u}$ is the generalized force of actuation
- $\mathbf{Q}_f$ is the generalized force modelling all friction forces in the system

Note that $\mathbf{G}(\mathbf{q}) = \mathbf{0}$, since the Otbot will move on flat terrain, and the constraint Jacobian $\mathbf{J}(\mathbf{q})$ was already obtained in Section 2.2.2. Thus, we only need to obtain $\mathbf{M}(\mathbf{q})$, $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$ and $\mathbf{E}(\mathbf{q})$ to set up the Lagrangian equation of motion.

We have already seen in Section 3.1 how to compute the mass matrix $\mathbf{M}(\mathbf{q})$. The Coriolis matrix can in turn be derived from the mass matrix using the following formula from [13], for each element (i,j) of $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$:

$$\mathbf{C}_{\mathrm{ij}} = \frac{1}{2}\sum_{\mathrm{k}=1}^{\mathrm{n}}\left(\frac{\partial\mathbf{M}_{\mathrm{ij}}}{\partial\mathbf{q}_{\mathrm{k}}} + \frac{\partial\mathbf{M}_{\mathrm{ik}}}{\partial\mathbf{q}_{\mathrm{j}}} - \frac{\partial\mathbf{M}_{\mathrm{kj}}}{\partial\mathbf{q}_{\mathrm{i}}}\right)\dot{\mathbf{q}}_{\mathrm{k}} \tag{3.2.2}$$

The resulting expression for this matrix is

$$\mathbf{C}(\mathbf{q},\dot{\mathbf{q}}) = \begin{bmatrix} 0 & 0 & \begin{matrix}\dot{\alpha}m_p(y_F s_\alpha - x_F c_\alpha)+ \\ (\dot{\alpha}-\dot{\varphi}_p)[m_b(y_G s_\theta - x_G c_\theta)+2l_1 m_w c_\theta]\end{matrix} & 0 & 0 & \begin{matrix}-(\dot{\alpha}-\dot{\varphi}_p)[2l_1 m_w c_\theta \\ +m_b(y_G s_\theta - x_G c_\theta)]\end{matrix} \\ 0 & 0 & \begin{matrix}-\dot{\alpha}m_p(y_F c_\alpha + x_F s_\alpha)+ \\ (\dot{\varphi}_p-\dot{\alpha})[m_b(y_G c_\theta + x_G s_\theta)-2l_1 m_w s_\theta]\end{matrix} & 0 & 0 & \begin{matrix}-(\dot{\alpha}-\dot{\varphi}_p)[2l_1 m_w s_\theta \\ -m_b(y_G c + x_G s_\theta)]\end{matrix} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

where once again s_θ and c_θ is a shorthand for $\sin\theta$ and $\cos\theta$, respectively.

Finally, since our vector $\mathbf{u}$ of motor torques is defined as $\mathbf{u} = [\tau_r, \tau_l, \tau_p]^T$ and each torque acts directly on a coordinate of $\dot{\boldsymbol{q}}$, the generalized force of actuation is $\boldsymbol{Q_a} = [0,0,0,\tau_r,\tau_l,\tau_p]^T$. Therefore, to rewrite this force in the form $\mathbf{Q}_a = \mathbf{E}\cdot\mathbf{u}$, we simply define

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

3.3 Conventional dynamics methods

We now wish to obtain the robot model in control form $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u})$, which requires writing $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$ as functions of $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\boldsymbol{u}$. To do so we will be using Lagrange's equation of motion in Eq. (3.2.1) 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 which requires the inversion of a 9×9 matrix.

3.3.1 Inverse dynamics

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

$$\mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}} + \mathbf{J}^T\,\boldsymbol{\lambda} = \mathbf{E}\,\mathbf{u}, \tag{3.3.1}$$

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. To solve this we define $\boldsymbol{\tau_{ID}} = \mathbf{M}\,\ddot{\mathbf{q}} + \mathbf{C}\,\dot{\mathbf{q}}$ and rewrite the previous equation as

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

or equivalently, as

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

It is easy to see that $[\mathbf{E} \;\; -\mathbf{J}^T]$ 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

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

which gives the solution for inverse dynamics as

$$\mathbf{u} = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{E} & -\mathbf{J}^T \end{bmatrix}^{-1} \boldsymbol{\tau_{ID}}. \tag{3.3.2}$$

3.3.2 Forward dynamics

The forward dynamics problem consists in finding the acceleration $\ddot{\mathbf{q}}$ that corresponds to a given action $\mathbf{u} = [\tau_r, \tau_l, \tau_p]^T$. Similarly to the inverse problem, we need to solve Eq. (3.3.1), but this time for $\ddot{\mathbf{q}}$, which gives us a system with 6 equations for 9 unknowns (6 coordinates in $\ddot{\mathbf{q}}$ and 3 in $\boldsymbol{\lambda}$). Thus, we need 3 additional equations to determine $\ddot{\mathbf{q}}$ and $\boldsymbol{\lambda}$ for a given $\mathbf{u}$. These equations are 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

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

This latter equation is called the acceleration constraint of the robot. By combining it with Eq. (3.3.1), we obtain a solvable linear system of 9 equations with 9 unknowns with the form:

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

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

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

which gives us a solution for the forward dynamics as follows:

$$\ddot{\mathbf{q}} = \begin{bmatrix} \mathbf{I}_6 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{M} & \mathbf{J}^T \\ \mathbf{J} & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{E}\,\mathbf{u} - \mathbf{C}\,\dot{\mathbf{q}} \\ -\dot{\mathbf{J}}\,\dot{\mathbf{q}} \end{bmatrix}. \tag{3.3.3}$$

3.4 Multiplier-free methods

The previous solutions to the inverse and forward dynamics (Eqs. (3.3.2) and (3.3.3)) require the inversion of a 6×6 and a 9×9 matrix respectively, which may be costly operations when we have to repeat them many times. We next see that, by exploiting two parameterizations of the feasible velocities $\dot{\boldsymbol{q}}$, we can find simpler solutions involving no matrix inverse, and a 6×6 matrix inverse only. These parameterizations are

$$\begin{bmatrix} \dot{\boldsymbol{p}} \\ \dot{\boldsymbol{\varphi}} \end{bmatrix} = \overbrace{\begin{bmatrix} \boldsymbol{I_3} \\ \boldsymbol{M_{IIK}} \end{bmatrix}}^{\boldsymbol{\Lambda}} \dot{\boldsymbol{p}} \quad \rightarrow \quad \dot{\boldsymbol{q}} = \boldsymbol{\Lambda} \cdot \dot{\boldsymbol{p}}, \tag{3.4.1}$$

$$\begin{bmatrix} \dot{\boldsymbol{p}} \\ \dot{\boldsymbol{\varphi}} \end{bmatrix} = \overbrace{\begin{bmatrix} \boldsymbol{M_{FIK}} \\ \boldsymbol{I_3} \end{bmatrix}}^{\boldsymbol{\Delta}} \dot{\boldsymbol{\varphi}} \quad \rightarrow \quad \dot{\boldsymbol{q}} = \boldsymbol{\Delta} \cdot \dot{\boldsymbol{\varphi}}, \tag{3.4.2}$$

where $\boldsymbol{M_{FIK}}$ and $\boldsymbol{M_{IIK}}$ are the matrices used to solve the instantaneous kinematic forward and inverse problems, respectively.

3.4.1 Inverse dynamics

Again, we wish to solve Eq. (3.2.1) for $\boldsymbol{u}$. To do this, we can multiply it by $\boldsymbol{\Delta}^T$ to obtain

$$\boldsymbol{\Delta}^T \boldsymbol{M} \ddot{\boldsymbol{q}} + \boldsymbol{\Delta}^T \boldsymbol{C} \dot{\boldsymbol{q}} + \boldsymbol{\Delta}^T \boldsymbol{J}^T \boldsymbol{\lambda} = \boldsymbol{\Delta}^T \boldsymbol{E} \cdot \boldsymbol{u} \tag{3.4.3}$$

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

$$\boldsymbol{\Delta}^T \boldsymbol{J}^T = 0. \tag{3.4.4}$$

This can be explained using the constraint in Eq. (2.2.1), which is satisfied only if $\dot{\boldsymbol{q}}$ is admissible. Since Eq. (3.4.2) also provides admissible $\dot{\boldsymbol{q}}$ for any $\dot{\boldsymbol{\varphi}}$, we have

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

Since the earlier result must hold for any $\dot{\boldsymbol{\varphi}}$, it must be that $\boldsymbol{J} \cdot \boldsymbol{\Delta} = 0$ and by taking the transpose we find that Eq. (3.4.4) must be true.

On the other hand, we can directly simplify the right-hand side of Eq. (3.4.3) since we have

$$\boldsymbol{\Delta}^T \boldsymbol{E} = \begin{bmatrix} \boldsymbol{M}_{FIK} & \boldsymbol{I}_3 \end{bmatrix} \begin{bmatrix} 0 \\ \boldsymbol{I}_3 \end{bmatrix} = \boldsymbol{I_3}. \tag{3.4.5}$$

Therefore, by combining Eq. (3.4.4) and Eq. (3.4.5) we obtain a closed-form expression for the inverse dynamics in which no matrix inverse is involved:

$$\boldsymbol{u} = \boldsymbol{\Delta}^T \boldsymbol{M} \ddot{\boldsymbol{q}} + \boldsymbol{\Delta}^T \boldsymbol{C} \dot{\boldsymbol{q}}. \tag{3.4.6}$$

3.4.2 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 $\boldsymbol{p}$ state variables. This can be done by using Eq. (3.4.1) and its derivative

$$\ddot{\boldsymbol{q}} = \boldsymbol{\Lambda} \ddot{\boldsymbol{p}} + \dot{\boldsymbol{\Lambda}} \dot{\boldsymbol{p}}. \tag{3.4.7}$$

By substituting them into Eq. (3.4.6), we find:

$$\begin{aligned} \boldsymbol{\Delta}^T \boldsymbol{M} (\boldsymbol{\Lambda} \ddot{\boldsymbol{p}} + \dot{\boldsymbol{\Lambda}} \dot{\boldsymbol{p}}) + \boldsymbol{\Delta}^T \boldsymbol{C} \boldsymbol{\Lambda} \dot{\boldsymbol{p}} &= \boldsymbol{u}, \\ \boldsymbol{\Delta}^T \boldsymbol{M} \boldsymbol{\Lambda} \ddot{\boldsymbol{p}} + \boldsymbol{\Delta}^T \boldsymbol{M} \dot{\boldsymbol{\Lambda}} \dot{\boldsymbol{p}} + \boldsymbol{\Delta}^T \boldsymbol{C} \boldsymbol{\Lambda} \dot{\boldsymbol{p}} &= \boldsymbol{u}, \\ \underbrace{\boldsymbol{\Delta}^T \boldsymbol{M} \boldsymbol{\Lambda}}_{\bar{\boldsymbol{M}}} \ddot{\boldsymbol{p}} + \underbrace{\boldsymbol{\Delta}^T (\boldsymbol{M} \dot{\boldsymbol{\Lambda}} + \boldsymbol{C} \boldsymbol{\Lambda})}_{\bar{\boldsymbol{C}}} \dot{\boldsymbol{p}} &= \boldsymbol{u}, \end{aligned}$$

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

$$\bar{\boldsymbol{M}} \ddot{\boldsymbol{p}} + \bar{\boldsymbol{C}} \dot{\boldsymbol{p}} = \boldsymbol{u}. \tag{3.4.8}$$

The earlier expression does not allow us to solve for $\ddot{\boldsymbol{q}}$ yet, but note that Eq. (3.4.7) can be expanded into:

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

whose second row gives us an expression of $\ddot{\boldsymbol{\varphi}}$ in terms of $\ddot{\boldsymbol{p}}$ and $\dot{\boldsymbol{p}}$. By combining this expression with Eq. (3.4.8) and rearranging the terms we obtain the system:

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

or in block form

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

which easily gives us the forward dynamics with the inversion of the 6×6 matrix on the right-hand side as

$$\ddot{\boldsymbol{q}} = \boldsymbol{K}^{-1} \cdot \boldsymbol{b}_{\boldsymbol{FD}}. \tag{3.4.9}$$

It is now simple to write the dynamical 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 $\mathbf{u}$ as defined in Section 2.1.2. To do this, we only need to augment Eq. (3.4.9) with the trivial equation $\dot{\mathbf{q}} = \dot{\mathbf{q}}$ to obtain

$$\begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}} \\ \begin{bmatrix} \bar{\boldsymbol{M}} & \mathbf{0} \\ -\boldsymbol{M}_{\boldsymbol{IIK}} & \boldsymbol{I}_3 \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{u} - \bar{\boldsymbol{C}}\dot{\boldsymbol{p}} \\ \dot{\boldsymbol{M}}_{IIK}\,\dot{\boldsymbol{p}} \end{bmatrix} \end{bmatrix}, \tag{3.4.10}$$

which gives the robot model in the usual control form

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

Chapter 4

Trajectory Optimization

Trajectory optimization can be defined as the process of designing a trajectory that minimizes some measure of performance while satisfying a set of constraints. It is typically used to compute an open-loop solution to an optimal control problem, therefore giving a solution as a set of control actions over time.

A trajectory optimization problem can take several forms, however we will focus here on single-phase continuous-time problems: those with smooth system dynamics throughout the trajectory, since this is the case for the Otbot dynamics in the studied scenarios. Moreover, we shall assume that the start and goal states of the robot, $\boldsymbol{x}_s$ and $\boldsymbol{x}_g$, are fixed and given for the trajectory to be obtained, though these could also be free if required, or loosely restricted by inequality or equality constraints.

Two different classes of methods can be used to solve a trajectory optimization problem, called *indirect* and *direct* methods [10, 11]. Although indirect methods provide high accuracy in the solution and assurance that it will satisfy the first order conditions of optimality, they typically have a small radii of convergence and require a very accurate initial guess. Direct methods, on the other hand, do not suffer from these disadvantages, but at the cost of providing often slightly less accurate solutions. Since we wish to plan reasonably good solutions from poor initial guesses, we will only be using direct methods in this thesis [12].

The principle behind direct methods is to discretize the continuous trajectory optimization problem in order to convert it into a constrained parameter optimization problem. This is called *transcription* and can be done through several methods, however we will focus here on *direct collocation* methods since they tend to be faster and solve a larger variety of problems. This discretized problem obtained via collocation is then solved numerically and the discrete solution can be interpolated to simulate the continuous-time state and control functions. This method for trajectory planning can be easily applied to optimize different parameters along the trajectory. Furthermore, it is extremely versatile and can be used to find optimal trajectories for many desired scenarios as we will see.

4.1 Problem formulation

Here will be presented a general form for continuous-time single-phase trajectory optimization problem, as well as a presentation of their implementation in the case of the Otbot.

To note, in optimization, the term *decision variable* is used to describe the variables that the optimization solver is adjusting to minimize the objective function. In our case, the decision variables are the state and control trajectories, $\mathbf{x}(t)$ and $\mathbf{u}(t)$, respectively. These are vectors which correspond to the ones described in Chapter 3.

A key component in any optimization problem is the objective function, which is the function to be minimized by the solver. This is the function that guides the optimization and its definition will completely change the nature of the found optimal solution. It can include two terms: a boundary objective $J(\cdot)$, called the Mayer term, and a path integral along the entire trajectory, with the integrand $w(\cdot)$, called the Lagrange term:

$$\min_{t_0, t_F, \boldsymbol{x}(t), \boldsymbol{u}(t)} \left(\underbrace{J\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right)}_{\text{Mayer Term}} + \underbrace{\int_{t_0}^{t_F} w(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau)) d\tau}_{\text{Lagrange Term}} \right). \tag{4.1.1}$$

A problem formulated using only the integral term is said to be in *Lagrange form*, whereas one using both terms is said to be in *Bolza form*, as seen in Eq. (4.1.1). Since any problem in Bolza form can be transformed into one in Lagrange form [14], we will only be using the Lagrange form here. The main two objective functions used with the Otbot are the force-squared and time objective function. They respectively minimize the torques used by the Otbot and the total time along the trajectory:

$$\textbf{Force-squared} \qquad w(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) = \boldsymbol{u}^T(t) \cdot \boldsymbol{u}(t). \tag{4.1.2}$$

$$\textbf{Time} \qquad w(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) = 1. \tag{4.1.3}$$

An optimization problem is subject to a set of constraints, which are detailed in Eq. (4.1.4)-(4.1.8). These constraints define the limits of the problem and its desired solution and are used to represent real-world constraints of the Otbot. This way, the computed optimal trajectories can be easily transposed to real-world situations, with little to no additional modification required.

The first constraint of the optimization problem is the system dynamics, which describes how the system evolves over time:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t, \boldsymbol{x}(t), \boldsymbol{u}(t)), \qquad \textbf{system dynamics.} \tag{4.1.4}$$

The complexity of the dynamics has a strong impact on the complexity of the optimization problem to be solved. With the Otbot, as seen in Chapter 3, the dynamics are non-linear.

Next is the path constraint, which enforces any constraints along the trajectory:

$$\boldsymbol{C_P}(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) \leq \mathbf{0}, \quad \textbf{path constraint.} \tag{4.1.5}$$

This allows a wide variety of possibilities to simulate any constraints on the state and control variables along the trajectory. In our case, it is used to enforce torque and state bounds, limit the maximum time of the trajectory and add obstacles to the problem to find collision-free trajectories, as detailed in Section 4.7. In a second time, they have also been used to implement velocity-dependent torque bounds to obtain trajectories compliant with more realistic motor capabilities, as seen in Section 4.9.

Next is the boundary constraint, which enforces any constraints on the initial and final states of the system:

$$\boldsymbol{C_B}\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right) \leq \mathbf{0}, \quad \textbf{boundary constraint.} \tag{4.1.6}$$

This is used with the Otbot to enforce bounds on the initial and final time and state to ensure the found trajectory starts and ends at $\boldsymbol{x}_s$ and $\boldsymbol{x}_g$. Since our problem is single-phased this constraint is only applied at the beginning and end of the trajectory.

There are often constant limits on the state or control variables, which we impose as bounds of the system. These are a specific type of path constraints that can be formulated as shown in Eq. (4.1.7). With the Otbot, these include boundaries on the state positions to represent walls which cannot be crossed. Constant limits on the control actions to limit the torques required by the motors are also imposed in this manner. Note that these inequalities are to be applied components wise:

$$\begin{aligned} \boldsymbol{x}_{\text{low}} &\leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}}\,, && \textbf{path bound on state,} \\ \boldsymbol{u}_{\text{low}} &\leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upp}}\,, && \textbf{path bound on control.} \end{aligned} \tag{4.1.7}$$

Finally, we have a specific set of boundary constraints for the initial and final time and state that can be formulated as shown in Eq. (4.1.8). They ensure the solution to a trajectory-planning method reaches the goal in a desired time window, which is crucial when not optimizing for time. As mentioned above, these are also used to define the initial and final desired states of the trajectory:

$$\begin{aligned} t_{\text{low}} &\leq t_0 < t_F \leq t_{\text{upp}}\,, && \textbf{bounds on initial and final time,} \\ \boldsymbol{x}_{0,\,\text{low}} &\leq \boldsymbol{x}\left(t_0\right) \leq \boldsymbol{x}_{0,\,\text{upp}}\,, && \textbf{bound on initial state,} \\ \boldsymbol{x}_{F,\,\text{low}} &\leq \boldsymbol{x}\left(t_F\right) \leq \boldsymbol{x}_{F,\,\text{upp}}\,, && \textbf{bound on final state.} \end{aligned} \tag{4.1.8}$$

This set of continuous-time equations are used to define the limits of the problem and the desired optimization. This formulation defines real-world system evolution and constraints as accurately as the provided dynamics, using decision variables as vector functions of time. This is a very accurate representation of the problem we wish to solve, however it is not numerically solvable in this form.

4.2 Problem transcription via collocation

To solve the previous problem numerically, we need to transcribe it into a constrained optimization problem formulated in terms of discrete states and actions. This transcription converts our trajectory optimization problem into a non-linear program, i.e. a constrained parameter optimization problem where the objective function and the constraints (including the system dynamics) are non-linear, which is the case for the Otbot. A typical formulation for a non-linear program is as follows:

$$\begin{aligned} \min_z J(\boldsymbol{z}) \quad & \text{subject to} \\ & \boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{0} \\ & \boldsymbol{g}(\boldsymbol{z}) \leq \boldsymbol{0} \\ & \boldsymbol{z}_{\text{low}} \leq \boldsymbol{z} \leq \boldsymbol{z}_{\text{upp}} \, . \end{aligned}$$

Transcription therefore entails a discretization of all continuous functions in the trajectory optimization problem. This will be done using collocation methods, which resort to polynomial splines to represent the state and action trajectories. A spline is a function made up of a sequence of polynomial segments, which are used here since they can be represented with a small set of coefficients and it is easy to compute derivatives or integrals of polynomial functions in terms of those coefficients.

Before discretizing the functions of the problem statement, however, we must first discretize the decision variables. This is done by representing the continuous state $\boldsymbol{x}(t)$ by $N+1$ values $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_N$ at specific points in time $t_0, \ldots, t_N$, which are known as *knot points*. A similar discretization is done to convert $\boldsymbol{u}(t)$ into a sequence of discrete actions $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_N$ at the mentioned knot points.

The number of knot points is one of the key parameters in direct collocation methods since it defines the number of segments in the spline approximation of the problem. A higher number of knot points usually implies a greater precision in the approximation of the problem at the cost of more computation. Note however that a high number of knot points does not necessarily correspond to a precise approximation of the continuous-time problem as this also depends on the type of collocation and more importantly the closeness between the type of the continuous problem and the approximation used.

Two direct collocation methods are used in this project: *Trapezoidal* collocation and *Hermite-Simpson* collocation, which respectively use the trapezoidal and Simpson quadrature rules. The trapezoidal method uses piecewise linear approximations, whereas the Hermite-Simpson gives approximations as piecewise quadratic functions. The second method is therefore more costly computationally but gives a solution that is higher-order accurate. The collocation method will also affect the number and placement of *collocation points*, which are the points in time used to compute the discrete approximation. For the trapezoidal method, these coincide exactly with the knot points. It is similar with the Hermite-Simpson method, with a third collocation point in the middle of each segment.

These integration approximations can be directly applied to integrals present in the trajectory optimization problem statement, such as the objective function in Lagrange form:

$$\textbf{Trapezoidal} \qquad \int_{t_0}^{t_F} w(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau))d\tau \approx \sum_{k=0}^{N-1} \frac{1}{2} h_k \cdot (w_k + w_{k+1}) \,.$$

$$\textbf{Hermite-Simpson} \qquad \int_{t_0}^{t_F} w(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau))d\tau \approx \sum_{k=0}^{N-1} \frac{h_k}{6} \left(w_k + 4w_{k+\frac{1}{2}} + w_{k+1} \right) .$$

These collocation methods are also applied to the continuous-time system dynamics to represent them as a set of constraints known as *collocation constraints*. This is done by writing the system dynamics in integral form and then approximating that integral using trapezoidal or Simpson quadrature. This approximation is then applied to every pair of knot points. Regarding the constant bounds of the problem, they are simply applied at every collocation point in the discrete problem. The same is done for the path constraints along the trajectory. For the boundary constraints, they are applied on the first and last knot points.

Once converted to a non-linear program form, our problem can be solved numerically by providing an initial guess. We then obtain a discrete optimal solution, which must be converted back to a continuous solution in order to be used for trajectory planning. To do this, we simply interpolate the discrete solution to a high number of points to simulate continuous behavior. It is crucial that the interpolation method corresponds to the collocation method to keep approximations consistent at each step.

4.3 Accuracy metrics

To ensure accuracy across the found trajectory, we must define metrics which evaluate the viability of the solution. To achieve this, three distinct metrics are computed and evaluated for each solution, as presented below.

The first and simplest accuracy metric is the kinematic error of the solution, which evaluates how well the trajectory satisfies the kinematic constraints of the system. In the Otbot system, these are derived from the rolling constraints as explained in Section 2.2 and can be expressed with Eq. (2.2.9). Since it has four scalar constraints, we have four distinct values for this metric. Since the values of these constraints must be zero to satisfy the no-sliding rule of the wheels of the Otbot, we can simply express the error as the value of each constraint for each time-step of the final trajectory. Thus, for a solution trajectory $\boldsymbol{x}(t)$, we define the kinematic error of $\boldsymbol{x}(t)$ as the vector function

$$\boldsymbol{\varepsilon_K}(t) = \boldsymbol{F}(\boldsymbol{x}(t)),$$

where $\boldsymbol{F}(\boldsymbol{x}(t))$ is the left hand side of Eq. (2.2.9). Although this metric is simple, it is key to obtaining feasible trajectories. Indeed, candidate solutions which violate kinematic constraints transgress the basis of the system model, and therefore will never result in feasible trajectories.

The next accuracy metric is an error estimate on how well the candidate trajectory satisfies the system dynamics between the knot points. As explained in Section 4.2, the trajectory optimization method enforces the continuous-time system dynamics as discrete collocation constraints along the candidate trajectory. Therefore, between each pair of knot points, the system dynamics are not enforced and we must check how well they are approximated in the found trajectory. This error will be called the dynamical error and is expressed as:

$$\boldsymbol{\varepsilon_D}(t) = \dot{\boldsymbol{x}}(t) - \boldsymbol{f}(t, \boldsymbol{x}(t), \boldsymbol{u}(t)),$$

where $\dot{\boldsymbol{x}}(t)$ is the interpolated dynamics from the candidate trajectory and $\boldsymbol{f}(t, \boldsymbol{x}(t), \boldsymbol{u}(t))$ are the system dynamics computed using the interpolated state and control variables from the same trajectory. This error will be zero at every collocation point and non-zero elsewhere.

Finally, the last accuracy metric evaluates how well the candidate trajectory corresponds to a 'real' trajectory at the knot points. To achieve this, we compute the 'real' trajectory using an ODE solver with the interpolated control of the candidate solution. Then for each state variable, we can compare it with the value of the candidate trajectory at each knot point. This metric is called the trajectory error and is closely linked to the dynamical error as they both show errors arising from the transcription. Nevertheless, both serve a purpose since the dynamical error shows an independent error for each segment of the candidate trajectory, whereas the trajectory error shows the accumulated sum of deviation at each knot point. The trajectory error is computed for each state variable as follows:

$$\varepsilon_{T,i}(t) = \left| x_i(t) - x_{real,i}(t) \right|.$$

Furthermore, the trajectory error is useful in the step following trajectory optimization, which is controlling the Otbot along the found optimal trajectory. The trajectory error shows how much the found trajectory deviates from the 'real' trajectory when using the same control. Thus, the trajectory error gives an insight as to show how much control effort will be necessary to keep the vehicle on the planned trajectory: additional control from computed optimal control which will be needed to compensate for approximation errors in trajectory optimization.

4.4 Trajectory optimization with the x model

Now that we know how to achieve trajectory optimization and how to evaluate the accuracy of the solution, we have the necessary tools to develop a trajectory optimizer for the Otbot. This optimizer has been implemented by using the trajectory optimization library for Matlab OptimTraj along with our previously-defined dynamical model. All results shown in this report were computed using Matlab R2020a, on a Windows 10(x64) MSI computer with an Intel Core i5-9300H CPU (2.40 GHz) and 16GB of RAM.

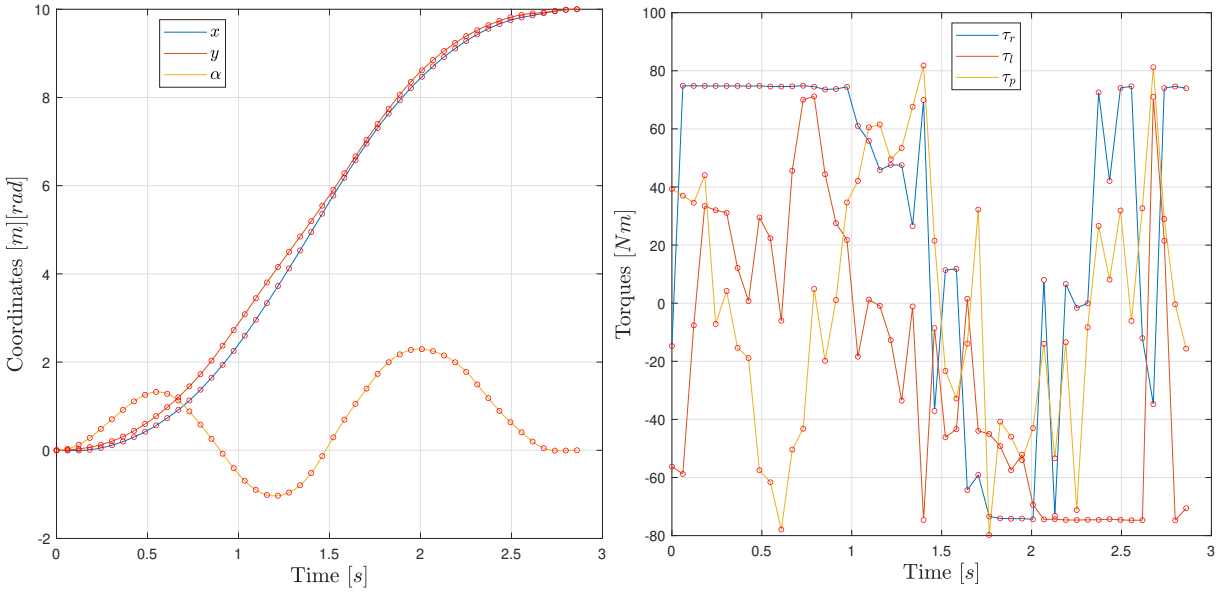
Table 4.1 provides the robot parameters assumed in all simulations, which correspond to a plausible prototype meant for load-carrying applications in factory environments. Initially, we shall assume our motors can apply the maximum torques provided at the end of the table, which are assumed to be constant. Later on we will show that the optimizer is also able to deal with velocity-dependent torque bounds however (see Section 4.9).

We can now define the tasks to be optimized. In order to evaluate the efficiency of a method and model, it is best to start by solving for simple tasks. This way, it is certain that any errors that arise in the found solution are not due to the complexity of the problem but to its formulation.

One of the simplest possible desired task is to go from a point A to a point B in a straight line with no obstacles. Therefore the basic problem formulation used here is to start the robot at $(x, y, \alpha) = (0, 0, 0)$ with zero velocity and end at $(x, y, \alpha) = (10, 10, 0)$ also with zero velocity. This task can and has been computed for different objective

Symbol	Meaning	Value	Unit
(x_G, y_G)	Coords. of G in the chassis frame F_c	(0,0)	$[m]$
(x_F, y_F)	Coords. of F in the platform frame F_p	(0,0)	$[m]$
m_b	Mass of the chassis base	105.0	$[kg]$
m_w	Mass of a wheel	2.0714	$[kg]$
m_p	Mass of the platform	21.94795	$[kg]$
I_b	Central moment of inertia of the chassis at G	1.06458	$[kg \cdot m^2]$
I_p	Central moment of inertia of the platform at F	2.22223	$[kg \cdot m^2]$
I_a	Axial moment of inertia of a wheel	1.03570*1e-2	$[kg \cdot m^2]$
I_t	Twisting moment of inertia of a wheel	5.61007*1e-3	$[kg \cdot m^2]$
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	$[m]$
$\tau_{max,w}$	Maximum torque for each wheel	75	$[Nm]$
$\tau_{max,p}$	Maximum torque for the pivot joint	230	$[Nm]$

Table 4.1 – Robot parameters and their values used in simulation



(a) $\boldsymbol{p}$-state profile

(b) Torques profile

Figure 4.1 – Trajectory and torque profiles for basic task using $\boldsymbol{x}$ model

functions, collocation methods and number of knot points. Shown in Fig 4.1 are the results when optimizing for time, using the trapezoidal collocation method with 48 knot points. The left plot provides the evolution of the x, y, and α coordinates along the optimized trajectory, and we clearly see that the robot is able to reach the goal position $(x, y, \alpha) = (10, 10, 0)$ for $t = 3$ s as expected. The right plot, in turn, shows the actions $\boldsymbol{u}(t)$ obtained. Note that since we are optimizing time, a bang-bang control is to be expected in this plot, so at least one of the motors should be applying its maximum or minimum torque at each time t of the trajectory. This is indeed the case in the initial and final parts of the trajectory, in which τ_r takes its maximum and minimum value, respectively. However, the transition from the former value to the latter is not sudden as it should be. This can be attributed to the fact that the trapezoidal rule is a low-order method that can only obtain approximate bang-bang solutions.

As can be seen in Fig. 4.2, the found solution here completely violates the kinematic constraints. This results in a completely unrealistic sliding behavior in the Otbot, as can be seen in videos here.

In Fig. 4.3, we can see the evolution of the dynamical error for the position state variables along the trajectory. In this and the following figures, the red dots represent the positions of the knot points, at which the dynamical error is 0. This is intrinsic to the collocation scheme, since the discrete collocation constraints are imposed at each collocation point, thereby enforcing the system dynamics. In between the knot points, we can see the error becomes significant.

As explained in Section 4.3, the kinematic constraint is key to finding a realistic solution. If it is not respected, the found solution cannot result in a feasible behavior.

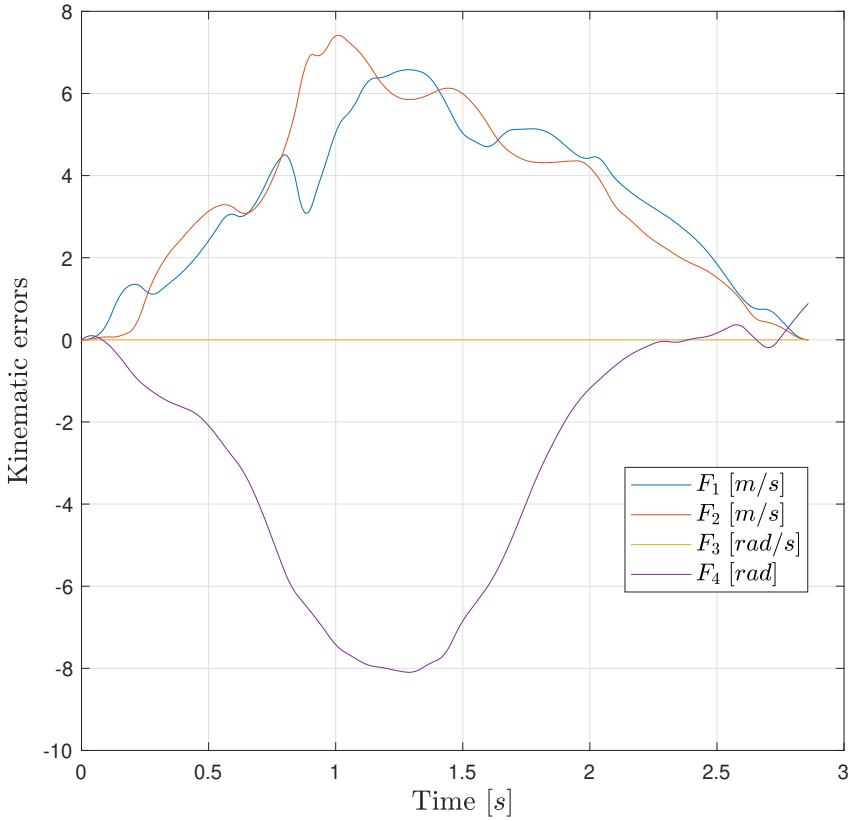


Figure 4.2 – Kinematic error for basic task using $\boldsymbol{x}$ model

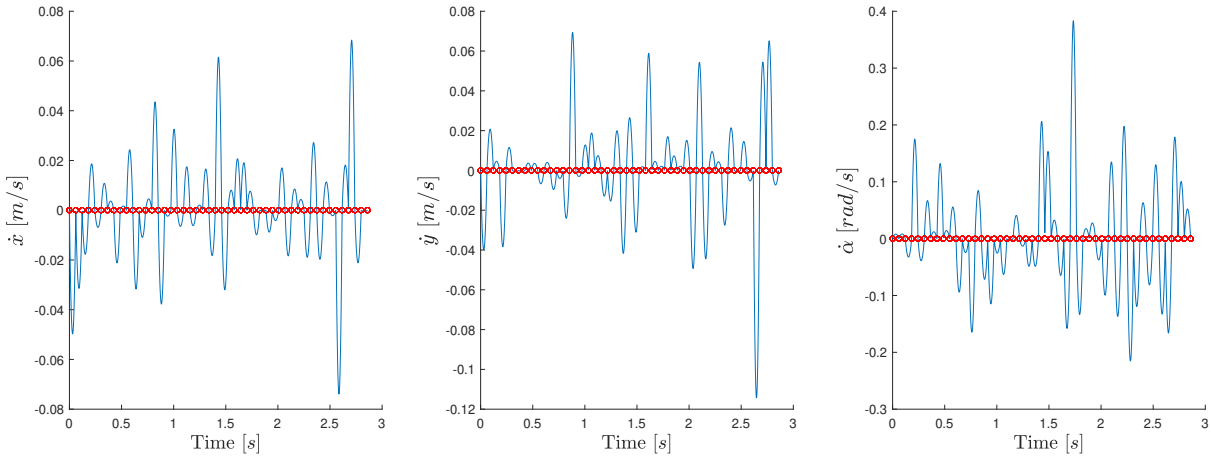


Figure 4.3 – Dynamical error for basic task using $\boldsymbol{x}$ model

Therefore, it is necessary to find solutions where the kinematic error is zero or very close to zero. To achieve this, several scenarios were tested by varying:

- the collocation method, either trapezoidal or using the Hermite Simpson rule
- the number of knot points: from 8 to 64

Unfortunately, every solution found using trajectory optimization techniques had a large kinematic error resulting in unfeasible behavior. This can be explained by the form

of our dynamical model and the resulting state-space in which it evolves. In fact, the state $\boldsymbol{x} = (\boldsymbol{q}, \dot{\boldsymbol{q}})$ is subject to our kinematic constraints in Eq. (2.2.9). Thus, the $\boldsymbol{x}$ coordinates are not independent since they must satisfy Eq. (2.2.9). The state-space on which the Otbot is constrained to move can therefore be defined as the set:

$$\mathcal{X} = \{\boldsymbol{x} : \boldsymbol{F}(\boldsymbol{x}) = 0\}, \tag{4.4.1}$$

where $\mathcal{X}$ is an 8-dimensional smooth manifold as we will see below. All feasible trajectories of the Otbot passing through a given initial state must lie on this manifold.

From the dependencies between variables in the $\boldsymbol{x}$ model, two problems arise during the trajectory optimization scheme. Firstly, these dependencies create redundant constraints at the boundary points of the trajectory, since the initial and final states must lie within an 8-dimensional manifold but the constraints specify values for the 12 coordinates of the state. These $12 - 8 = 4$ redundant constraints either make the problem unsolvable (if the start or end states do not satisfy the kinematic constraints) or they make it impossible to apply the Karush-Khun-Tucker conditions for optimality, since the constraints do not fulfill the linear independence constraint qualification [15]. Secondly, these dependencies create a cumulative drift away from $\mathcal{X}$ due to the approximation that occurs during the transcription process. Small errors due to approximating our system dynamics will accumulate along the computed discrete trajectory (Fig. 4.4(a)). This resulting trajectory therefore contains unfeasible states which do not respect the imposed kinematic constraints.

In the following section we will remedy this problem by defining a new dynamic model that uses only a subset $\boldsymbol{z}$ of the variables in $\boldsymbol{x}$, which are independent among themselves. By solving our trajectory optimization problem in the $\boldsymbol{z}$ space, and then reconstructing our trajectory on the manifold $\mathcal{X}$, we will be able to fully eliminate the kinematic error by construction (Fig. 4.4(b)).

4.5 A reduced model to avoid drift

We wish to design a reduced dynamical model with independent state variables. For this, we must be able to express some variables present in

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

as functions of the remaining variables. This is simple for the $\dot{\boldsymbol{\varphi}}$ variables, which can be expressed as a function of $\dot{\mathbf{p}} = (\dot{x}, \dot{y}, \dot{\alpha})$ using the solution for the inverse instantaneous kinematic problem:

$$\dot{\boldsymbol{\varphi}} = \mathbf{M}_{IIK}(\mathbf{q}) \cdot \dot{\mathbf{p}}. \tag{4.5.1}$$

Furthermore, by using the previously-defined holonomic constraint in Eq. (2.2.8), we can express one of the four angles in our state coordinates as a function of the other

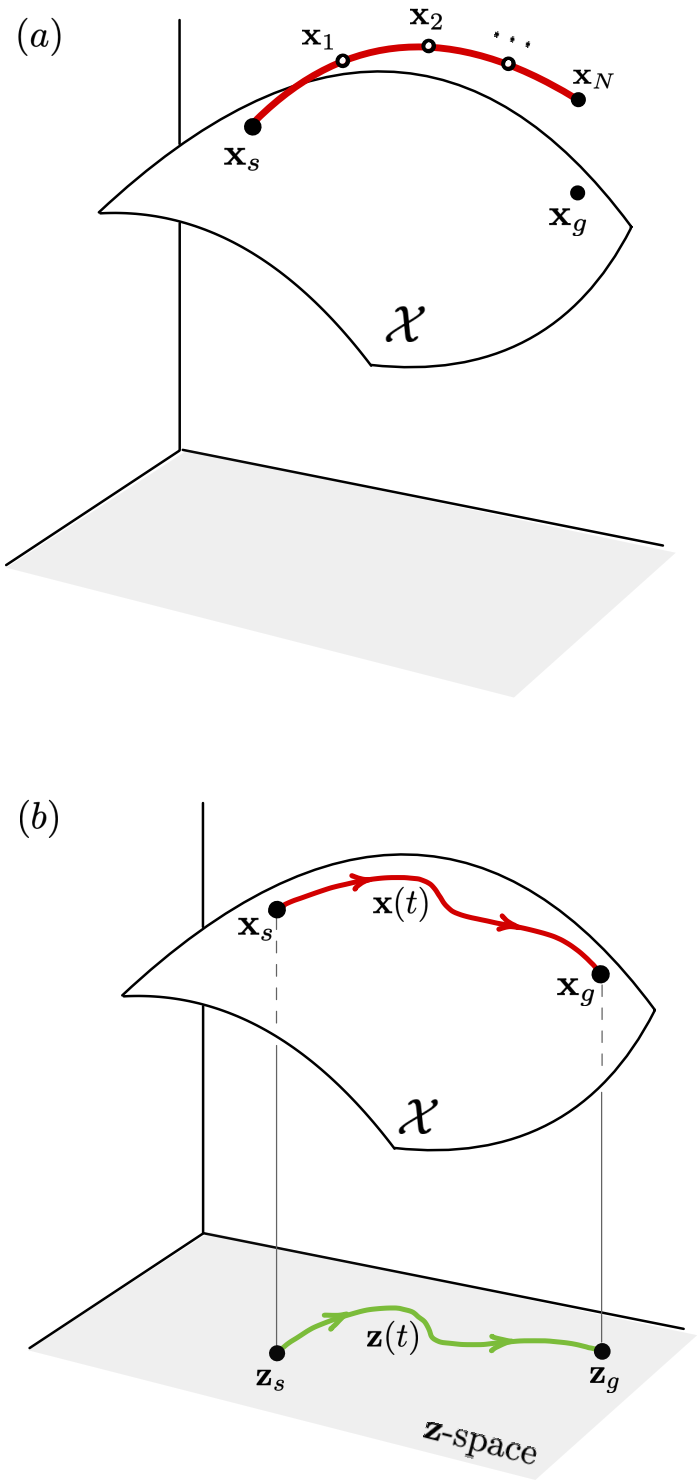


Figure 4.4 – (a) Because of the dependence among the $\boldsymbol{x}$ variables, the trajectory $\boldsymbol{x}(t)$ drifts away from $\mathcal{X}$ if we solve the trajectory optimization problem using $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$. (b) To avoid this problem, we write the system dynamics using a subset $\boldsymbol{z}$ of independent state coordinates from $\boldsymbol{x}$ and solve the optimization problem in $\boldsymbol{z}$ space. We then recover $\boldsymbol{x}(t)$ on $\mathcal{X}$ from $\boldsymbol{z}(t)$ by using the parameterization of $\mathcal{X}$ induced by the $\boldsymbol{z}$ coordinates.

three. Indeed, by isolating φ_l from Eq. (2.2.8), we can easily get

$$\varphi_l = -\frac{2\ell_2}{r}\alpha + \varphi_r + \frac{2l_2}{r}\varphi_p - \frac{2l_2}{r}K_0. \tag{4.5.2}$$

Therefore, using (4.5.2) and (4.5.1), we can compute a global parametrization of $\mathcal{X}$ with the form

$$\boldsymbol{x} = \boldsymbol{\Psi_p}(\boldsymbol{z}),$$

where

$$\begin{aligned}\boldsymbol{x} &= (x, y, \alpha, \varphi_r, \varphi_l, \varphi_p, \dot{x}, \dot{y}, \dot{\alpha}, \dot{\varphi}_r, \dot{\varphi}_l, \dot{\varphi}_p),\\ \boldsymbol{z} &= (x, y, \alpha, \dot{x}, \dot{y}, \dot{\alpha}, \varphi_r, \varphi_p),\end{aligned}$$

and

$$\boldsymbol{\Psi_p}(\boldsymbol{z}) = \begin{bmatrix} x \\ y \\ \alpha \\ \varphi_r \\ -\frac{2\ell_2}{r}\alpha + \varphi_r + \frac{2l_2}{r}\varphi_p - \frac{2l_2}{r}K_0 \\ \varphi_p \\ \dot{x} \\ \dot{y} \\ \dot{\alpha} \\ \boldsymbol{M_{IIK}} \cdot \dot{\boldsymbol{p}} \end{bmatrix}. \tag{4.5.3}$$

Since the Jacobian $\dfrac{\partial \boldsymbol{\Psi_p}}{\partial \boldsymbol{z}}(\boldsymbol{z})$ is full rank for all $\boldsymbol{z} \in \Re^8$, this implies $\mathcal{X}$ is a smooth manifold of dimension 8.

The inverse map of this parametrization is:

$$\boldsymbol{z} = \boldsymbol{\Psi_c}(\boldsymbol{x}) = \boldsymbol{L} \cdot \boldsymbol{x}, \tag{4.5.4}$$

where $\boldsymbol{L}$ is a constant sparse matrix of zeroes and ones simply selecting the $\boldsymbol{z}$ variables from $\boldsymbol{x}$.

Using these maps, we can now write the dynamic model of the Otbot using the $\mathbf{z}$ coordinates only:

$$\begin{aligned}\dot{\boldsymbol{z}} &= \boldsymbol{L} \cdot \dot{\boldsymbol{x}},\\ \dot{\boldsymbol{z}} &= \boldsymbol{L} \cdot \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}),\\ \dot{\boldsymbol{z}} &= \boldsymbol{L} \cdot \boldsymbol{f}(\Psi_p(\boldsymbol{z}), \boldsymbol{u}).\end{aligned} \tag{4.5.5}$$

Hereafter, this model will be written as

$$\dot{\boldsymbol{z}} = \boldsymbol{g}(\boldsymbol{z}, \boldsymbol{u}),$$

where $\boldsymbol{g}(\boldsymbol{z}, \boldsymbol{u})$ refers to the right-hand side of Eq. (4.5.5).

4.6 Trajectory optimization with the z model

Since this parametrization uses independent variables, we can use it for trajectory optimization and avoid the problems caused by using the $\boldsymbol{x}$ model. We can easily transform the trajectory optimization formulation to use $\boldsymbol{z}$ coordinates and thus obtain our problem to be solved as:

$$\min_{t_F, \boldsymbol{z}(t), \boldsymbol{u}(t)} \int_{t_0}^{t_F} \omega(\tau, \boldsymbol{z}(\tau), \boldsymbol{u}(\tau)) d\tau,$$

subject to the constraints:

$$\begin{aligned} &\dot{\boldsymbol{z}}(t) = \boldsymbol{g}(t, \boldsymbol{z}(t), \boldsymbol{u}(t)), && \textbf{system dynamics.} \\ &\boldsymbol{C_P}(t, \boldsymbol{z}(t), \boldsymbol{u}(t)) \leq 0, && \textbf{path constraints.} \\ &\boldsymbol{C_B}\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right) \leq 0, && \textbf{boundary constraints.} \end{aligned}$$

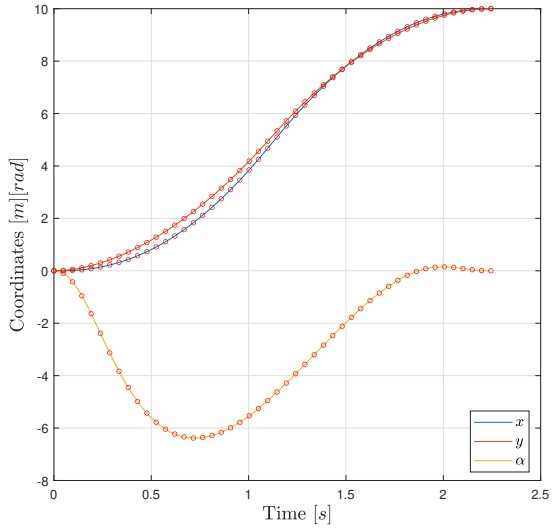
Now to compare the $\boldsymbol{z}$ model with the same basic task as presented in Section 4.4: from $(x, y, \alpha) = (0, 0, 0)$ to $(x, y, \alpha) = (10, 10, 0)$, optimizing for time, using the trapezoidal collocation method with 48 knot points. Videos are available here.

Firstly, the parametrization contains the kinematic constraints implicitly, and therefore cannot violate any of these constraints during the problem transcription. The kinematic error here is of the order of 10^{-14} and thus very close to zero, as with all trajectories using the $\boldsymbol{z}$ model. We will therefore now omit this metric when presenting further results.

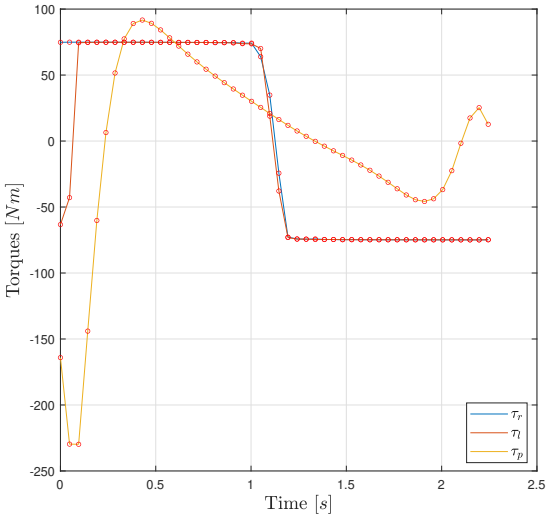
Concerning the dynamical error, when comparing with the $\boldsymbol{x}$ model in Fig. 4.3 it is evident the $\boldsymbol{z}$ model provides a drastic improvement with regards to respecting the system dynamics. Indeed, the maximum dynamical error is lower and it is only present at certain points of the trajectory. Similarly, the trajectory error is quite small, 2 cm for a 10 m trajectory, which indicates a low additional control effort. Furthermore, the time taken to reach the target has also decreased, most likely due to the more constant control actions. In fact, we can see in Fig. 4.5 that the torque profile follows a bang-bang curve almost perfectly, using either the maximum or minimum torque available. This is the optimal behavior when optimizing for time since to reach a target quickly, the Otbot must first accelerate as fast as possible before breaking as hard as possible.

Note that the trajectory includes some movement of the platform even though it has the same start and end position. This movement is completely allowed by the problem formulation and appears to help the Otbot by creating a small counter torque. Nevertheless this is seemingly unnecessary behavior and could become a problem when transporting items on the platform. Fortunately, it can be easily avoided by adding to the time objective function a term penalizing the pivot torque employed as shown in Eq. (4.6.1), where $w(\cdot)$ is the term integrated along the entire trajectory and c a coefficient to vary the importance of minimizing the torque over minimizing time.

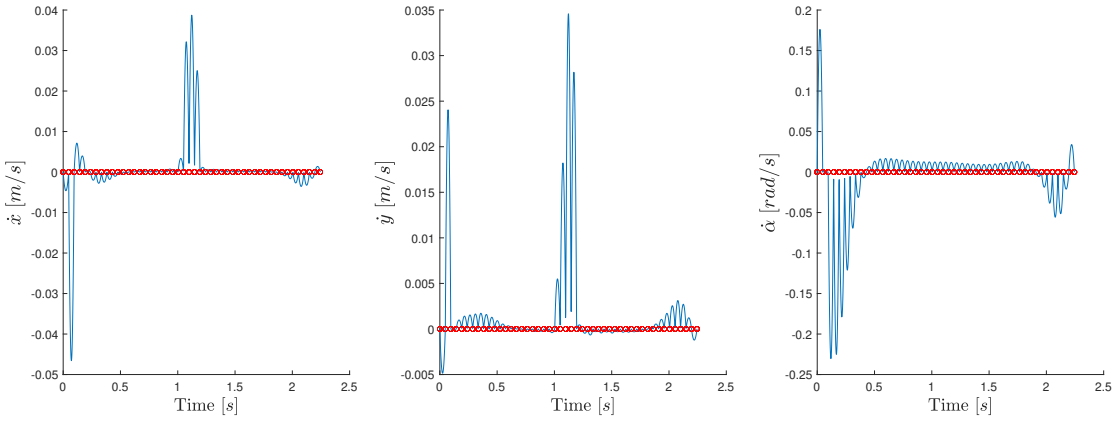
$$\textbf{Time + Pivot torque} \qquad w(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) = (1 - c) + c \cdot \tau_p^2. \tag{4.6.1}$$



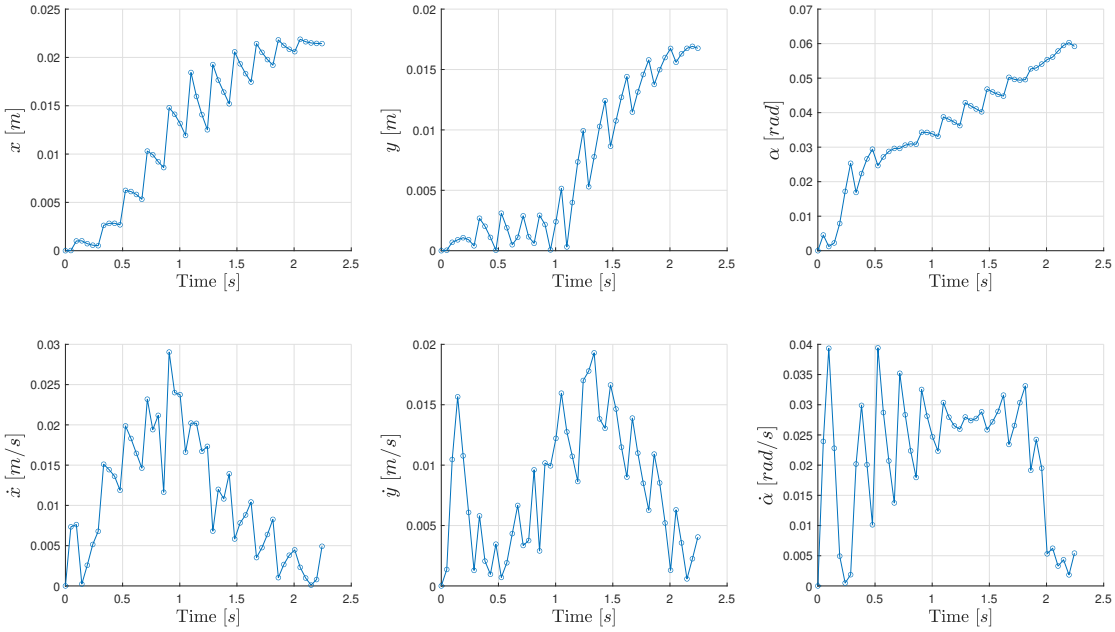
(a) $\boldsymbol{p}$-state profile



(b) Torques profile

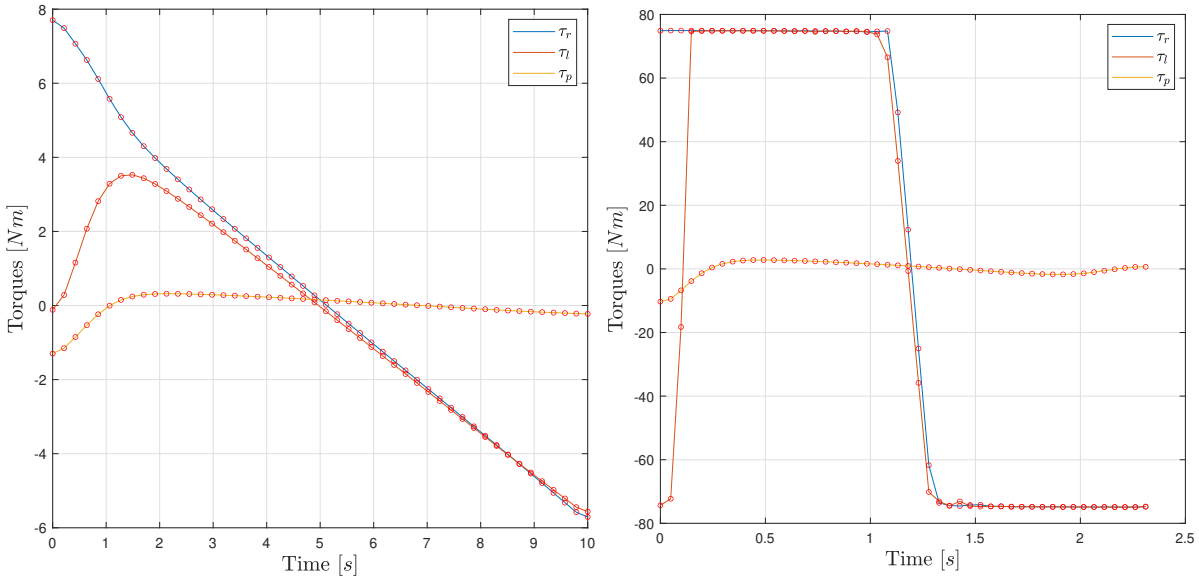


(c) Dynamical Error



(d) Trajectory Error

Figure 4.5 – Trajectory, torque and error profiles for basic task using $\boldsymbol{z}$ model



(a) Minimizing force-squared

(b) Minimizing time and τ_p

Figure 4.6 – Torque profiles for basic task with other objective functions

As explained in Section 4.2, different objective functions can be chosen to minimize different parameters along the trajectory. Shown in Fig. 4.6 are two torque profiles of the same basic task with objective functions minimizing either the force-squared or time and the pivot torque squared.

The force-squared objective function is useful to reduce the control effort of the robot, since the required torques will be kept low during the trajectory. However, this strategy will provide solutions which take up the entire available time, resulting in slow, low-effort trajectories. Here, a maximum time of 10s has been imposed as a boundary constraint, so the found trajectory reaches the target position in 10s.

The time and pivot torque objective function as described above is used to reduce the movement of the platform whilst minimizing for time. Comparing with Fig. 4.5 we can see the pivot torque is drastically reduced with only a slight increase in the time taken to reach the goal position. Here the torque coefficient c has a value of 0.0001.

Two other key parameters we can vary during trajectory optimization are the collocation method and the number of knot points. Fig. 4.7 offers a comparison of the dynamical error and computation time for trapezoidal and Hermite-Simpson collocation methods for several numbers of knot points when solving for the same basic task as before. It is important to keep in mind that any observations drawn from these graphs are only relevant to this particular task and does not necessarily demonstrate the full extent of using different collocation methods.

Nevertheless, it is easy to see here that the general rule concerning the number of knot points holds: increasing the number of points reduces the errors and increases computation time. More precisely, we see that after reaching a certain number of points,

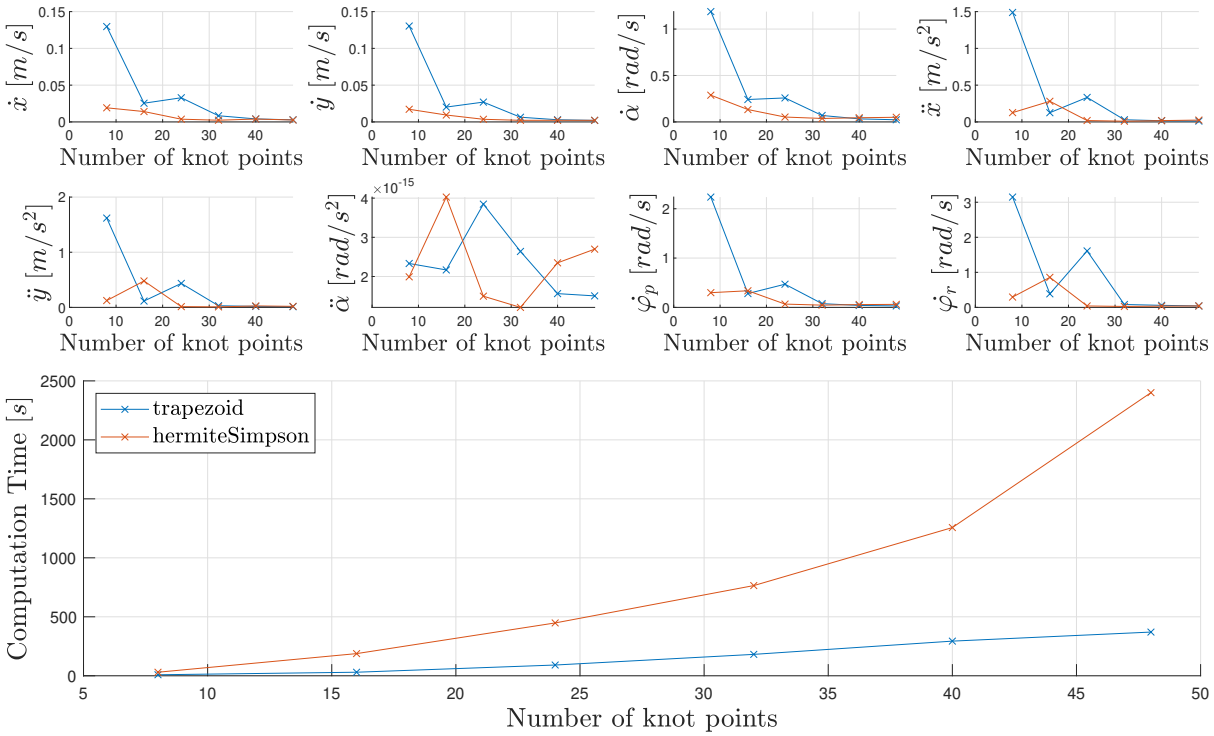


Figure 4.7 – Average dynamical error (top) and computation time (bottom) for different collocation methods and number of knot points with time objective function

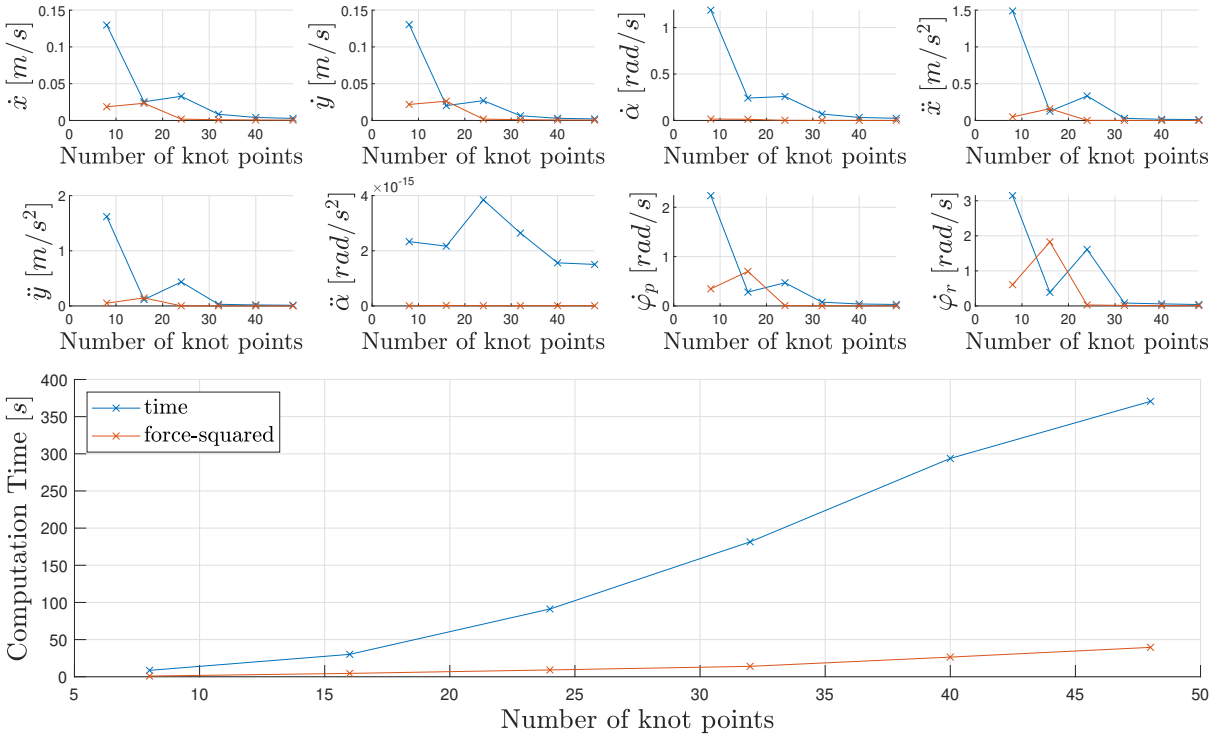


Figure 4.8 – Average dynamical error (top) and computation time (bottom) for different objective function and number of knot points using the trapezoidal collocation method

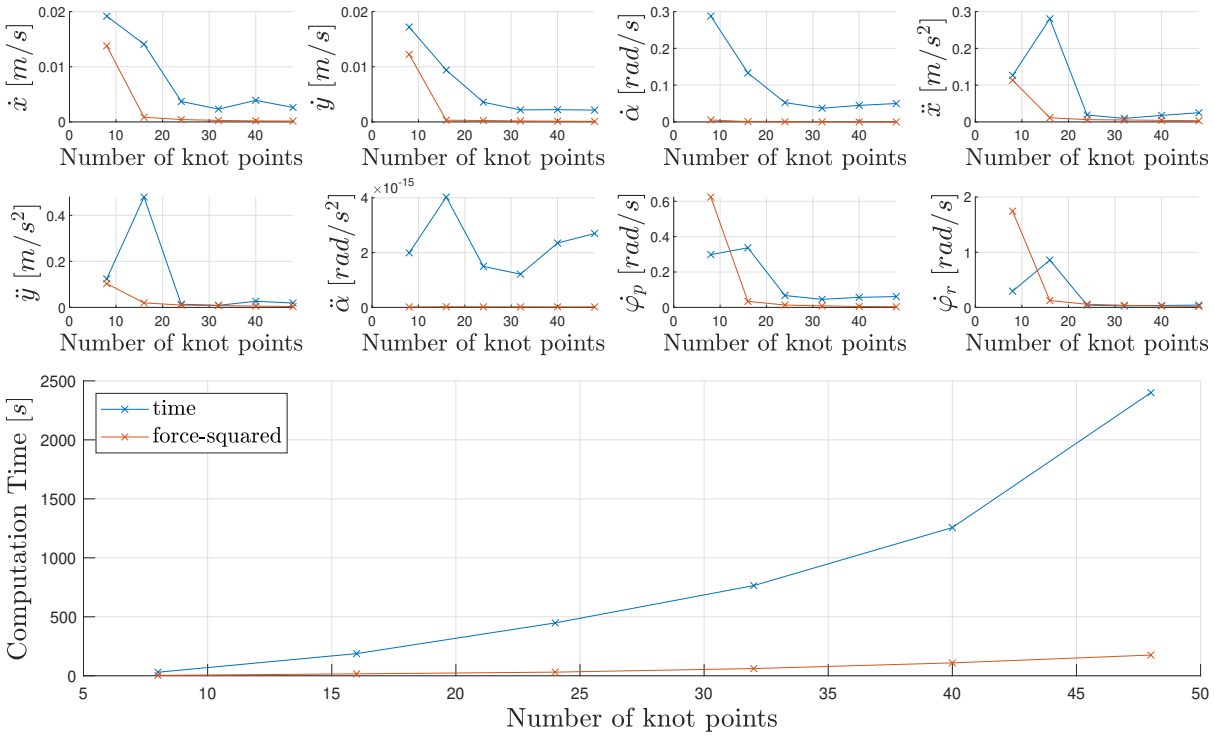


Figure 4.9 – Average dynamical error (top) and computation time (bottom) for different objective functions and number of knot points using the Hermite-Simpson collocation method

the error barely decreases as it is already very close to zero. However the computation time keeps increasing non-linearly, especially for the Hermite-Simpson method. This is due to the third collocation point used in this method which means that for the same number of knot points, the Hermite-Simpson method is at least 50% more costly than the trapezoidal method. This shows how for simple trajectories, there is no particular advantage to using high-order collocation methods or a very high number of knot points. Indeed this will only increase computation time with almost no improvement in the found solution.

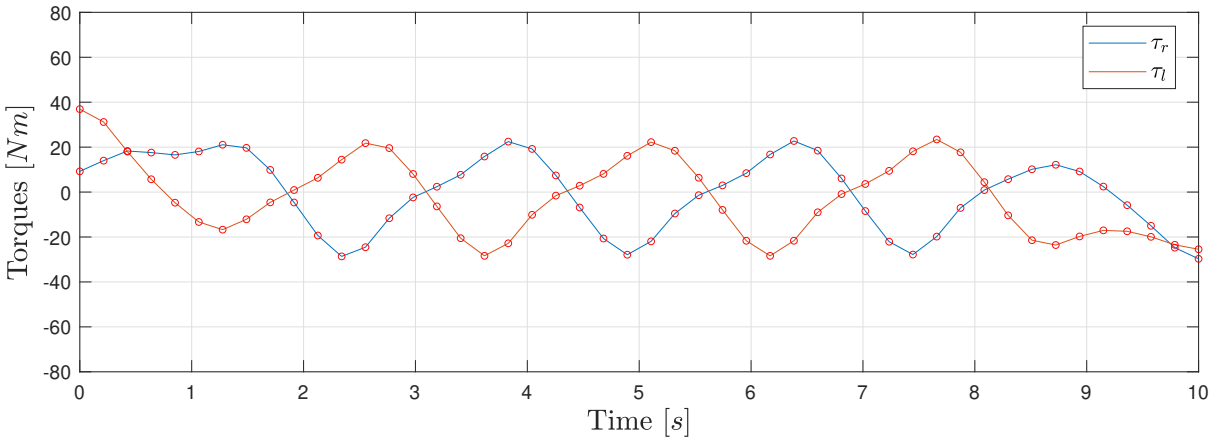
Similarly, Figs. 4.8 and 4.9 provide those values this time comparing objective functions for each collocation method.

It is also interesting to compare the torque profile for different objective functions in a more complex scenario than a straight line. With this scenario, presented in Section 4.7, we impose deviations from a straight line by adding obstacles which the robot must avoid. With these deviations, we can observe more changes in torques and so better see the impact of each objective function. We compare the force-squared and time objective functions presented previously, as well as the force-derivative one shown here:

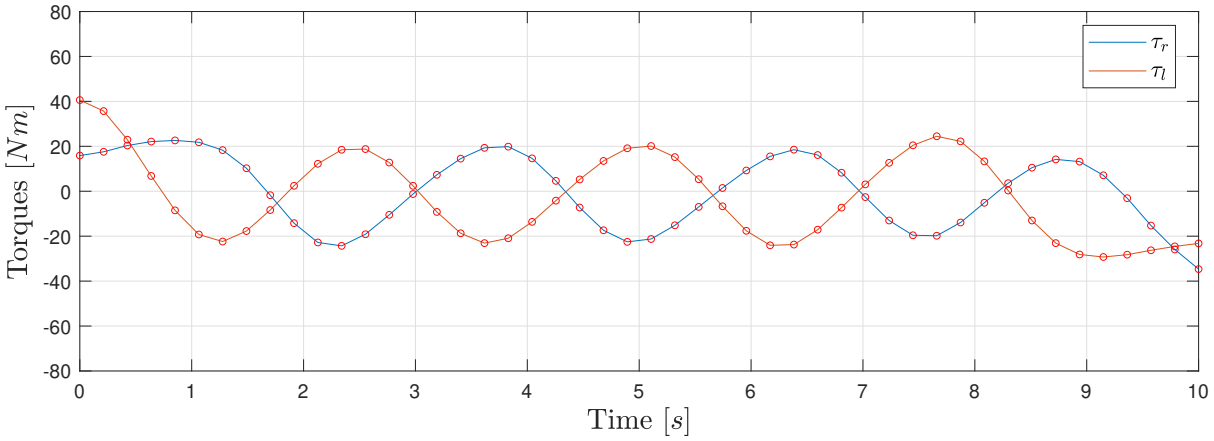
$$\textbf{Force-derivative} \qquad w(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) = \dot{\boldsymbol{u}}^T(t) \cdot \dot{\boldsymbol{u}}(t). \tag{4.6.2}$$

This last function, which minimizes the derivative of the torques imposes an even smoother torque profile than minimizing directly the torques, as seen in Fig. 4.10b. In contrast,

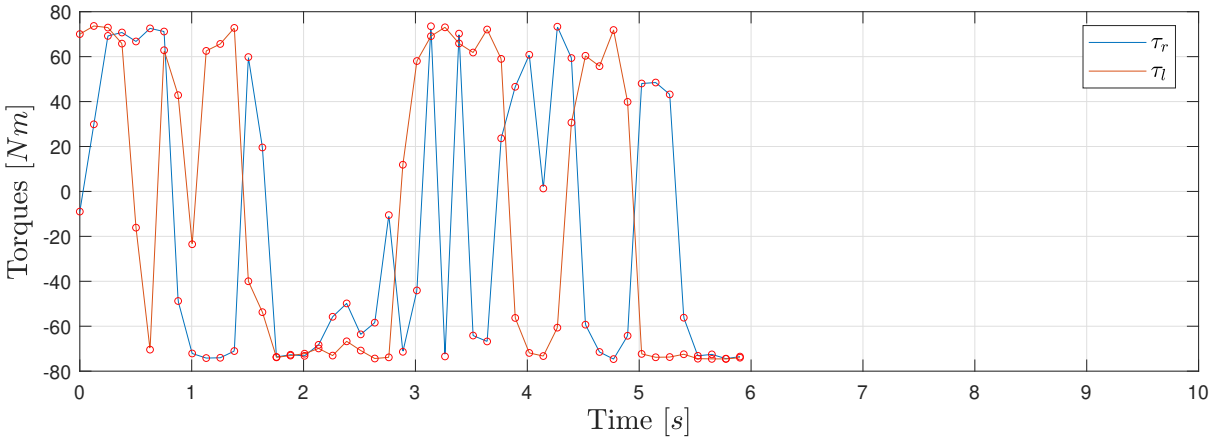
in the case of minimizing time in Fig. 4.10c, we can see a profile closer to a bang-bang trajectory.



(a) Force-squared objective function



(b) Force-derivative objective function



(c) Time objective function

Figure 4.10 – Torque profile comparison for different objective functions in more complex scenario using trapezoidal method with 48 knot points

4.7 Obstacle avoidance

Using the path constraints in the trajectory optimization problem formulation, it is possible to add round obstacles which the robot must avoid. We will present how to model such obstacles, several examples of scenarios involving obstacles, as well as some methods to ensure a feasible solution is found.

In a trajectory optimization problem, obstacles are formulated by modeling the Otbot as a circle and constraining the distance between the center of Otbot and the center of the obstacle to be greater than the sum of their radii. The general expression of a path constraint as given in Eq. (4.1.5) consists in an inequality of the form $h \leq 0$, which can be achieved by using the Euclidean distance as follows:

$$\begin{aligned}
\textbf{Euclidean distance} \quad & d(o,r)^2 = (o_x - r_x)^2 + (o_y - r_y)^2 . \\
\textbf{Distance constraint} \quad & d(o,r) \geq o_r + r_r . \\
\textbf{Combination} \quad & (o_x - r_x)^2 + (o_y - r_y)^2 \geq (o_r + r_r)^2 . \\
\textbf{Path constraint} \quad & (o_r + r_r)^2 - \left((o_x - r_x)^2 + (o_y - r_y)^2\right) \leq 0, \qquad (4.7.1)
\end{aligned}$$

where o and r represent the obstacle and robot, respectively. Subindices x, y and r denote the x and y coordinates of their centers and their radius, respectively. Therefore to include round obstacles, we simply need the position of their center and their radius. The radius of the robot r_r, is set to 0.5 meters to provide a reasonable clearance in the task of obstacle avoidance.

Using Eq. (4.7.1), we can model any number of obstacles into the trajectory optimization scheme. Each obstacle will be computed as a distinct path constraint to be evaluated at each collocation point, therefore adding many obstacles can greatly increase computations. A first example scenario can be seen in Fig. 4.11 where the Otbot must cross a crowded corridor.

This is a simple obstacle avoidance scenario but we must be careful when setting the collocation method and number of knot points to be used. Since the path constraints are only imposed at the collocation points, the robot can violate those constraints in between two points. Therefore it is crucial to have a high-enough number of knot points

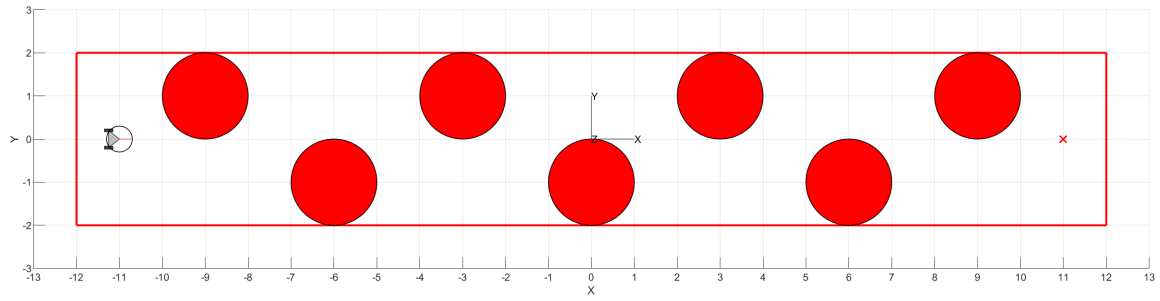


Figure 4.11 – Crowded corridor scenario

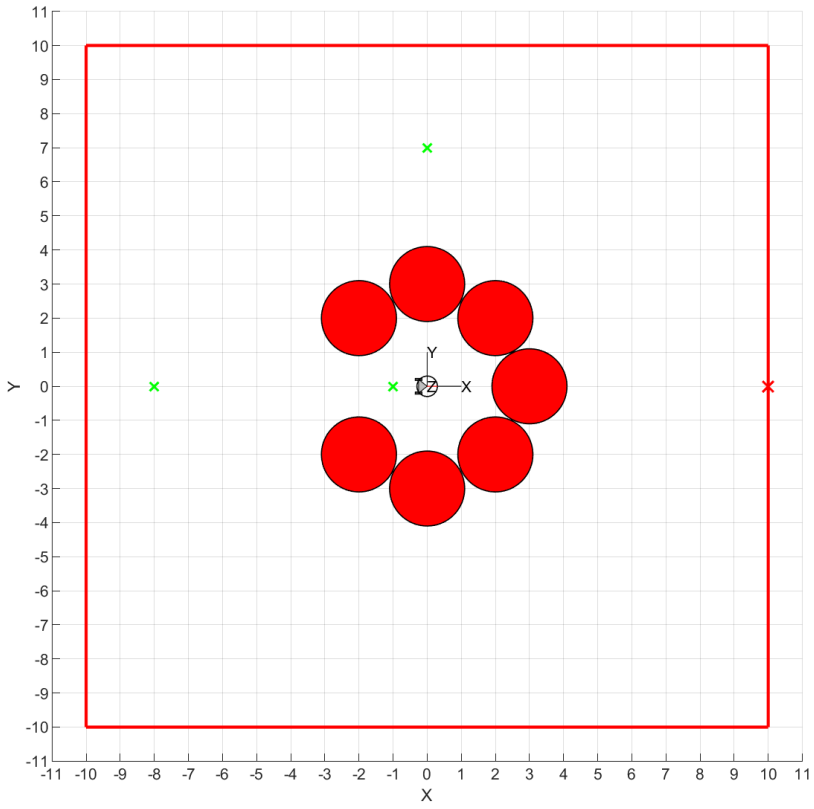
to eliminate the possibility of having an inter-knot period in which the constraint can be violated so that the obtained trajectory truly avoids obstacles. Too few knot points will often result in the robot accelerating through obstacles at full speed in a manner which respects the respects the obstacle constraints at the collocation points but not between them.

The results presented in this section are obtained using the minimum number of knot points that respected obstacle constraints. This is the best trade-off between computation time and the imposed constraints. Computing solutions with a higher number of knot points would further reduce the dynamical error but not significantly improve the solution in terms of obstacle avoidance and in return would have a high computational cost. The best computed solutions for all obstacles avoidance scenarios are available in this playlist, starting with the first presented scenario. The specifics of the parameters used to compute each solution are available in the description of each video, and plots of the state, torques and errors are available in the Appendix B.

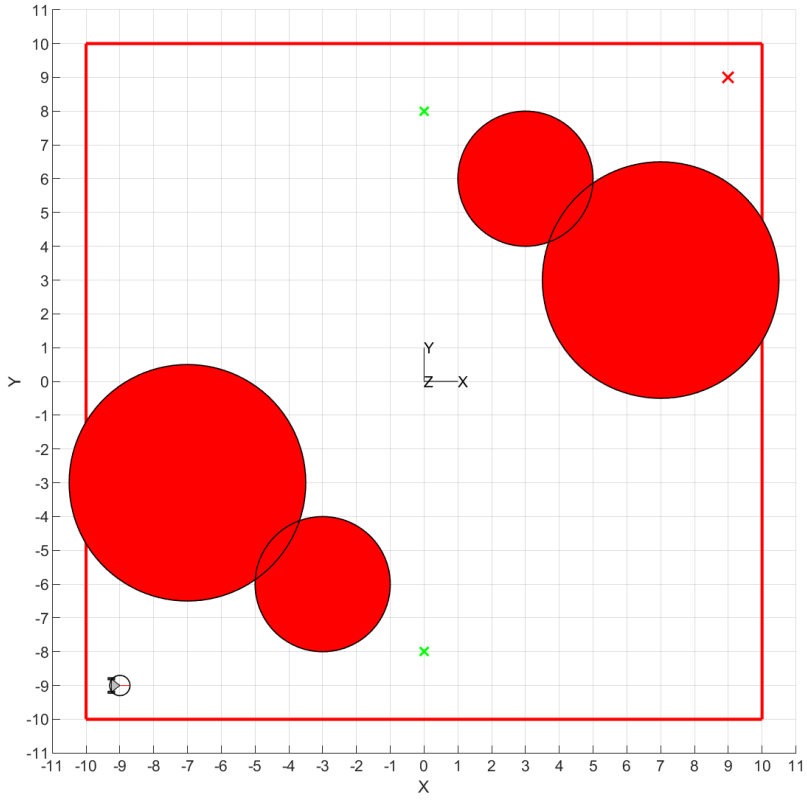
More complex problems can be created when adding obstacles. For example, we can force a non-linear trajectory. This creates some complications since feasible trajectories are no longer a straight line between two points, or a minor deviation of that, as has been the case up until now. To resolve this, we use a vital parameter of trajectory optimization which has not been discussed yet: the *initial guess*. This is the initial solution provided when solving our discrete constrained optimization problem, around which the solver searches for better feasible solutions. It includes the state, control and corresponding time for a number of points along a trajectory. It is desirable that this guess is a feasible trajectory in order to favor the convergence of the solver.

For the simple scenarios shown previously, this initial guess can be extremely simple and has little impact on the resulting solution. In fact the one used until now was only made up of the start and end positions for a given scenario, which the solver interpolates to find intermediate initial values. This has worked well since resulting trajectory are close to a straight line between the two points. However for more complex problems, where a straight line is not a feasible solution, it is necessary to give a more complete initial guess for the solver to converge on a feasible solution.

Presented in Fig. 4.12 are scenarios which force trajectories that are far from linear. As such, the initial guess provided is composed of the start and end positions and a few extra points marked in green, which will be called the *simple initial guess*. The scenarios here are solved using the force-squared objective function since it converges to an optimal solution faster and outputs smooth trajectories with low dynamical error. If one wished to obtain a faster solution, we could solve the same problems whilst minimizing for time. Doing this with the simple initial guess however results in solutions with significant dynamical errors. A better alternative is to provide the optimal force-squared trajectory as the initial guess and minimize time for this trajectory. This is much faster and provides good results.



(a) Bug trap problem



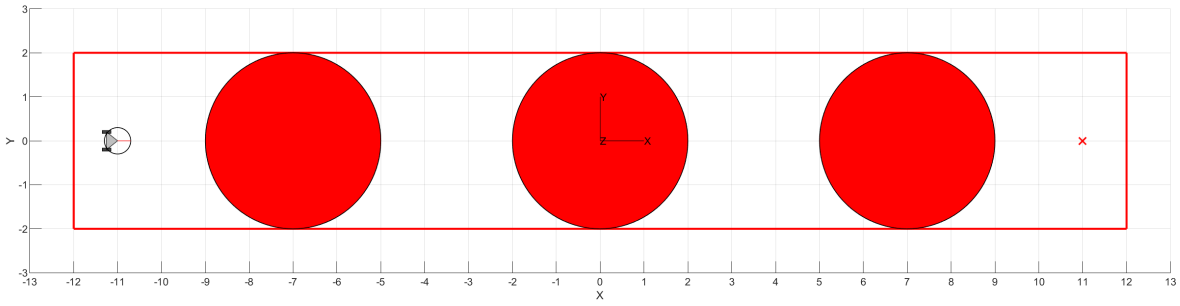
(b) Forced deviation problem

Figure 4.12 – Obstacle avoidance scenarios with static obstacles

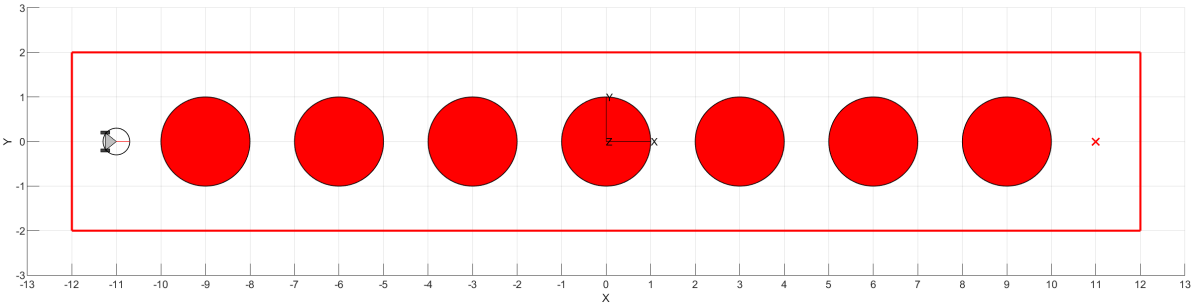
This technique is similar to *mesh refinement*, which is trajectory optimization with several episodes, starting form a low-order method with few knot points and gradually increasing the number of knot points or method order, each time providing the previous found solution as the initial guess. This method is particularly useful to reduce dynamical errors, especially in complex scenarios.

With the trajectory optimization problem formulation, it is also possible to model moving obstacles. To do this, we simply augment Eq. (4.7.1) with a time component so that the position of the center of the obstacles is a function of time rather than a fixed point.

The first scenario implemented here is similar to the crowded corridor example, this time with moving obstacles. Several similar scenarios were tested, keeping the same bounds and goal but changing the number of obstacles or their speed. A snapshot of the first and last scenarios can be seen in Fig. 4.13, where the obstacles will be moving up and down perpendicularly to the Otbot trajectory. In this case, a basic initial guess with only the start and end position can be used since the solution only has small deviations from the straight line between these two points. Even if it involves moving obstacles, solving this problem is not more costly computationally, as the computational cost depends on the number of obstacles and not their movement. In fact from a trajectory optimization perspective, this problem is easier to solve than the bug trap problem since it is more a question of timing in input variation rather than path planning.



(a) Crowded corridor with 3 moving obstacles



(b) Crowded corridor with 7 moving obstacles

Figure 4.13 – Obstacle avoidance Scenarios with moving obstacles

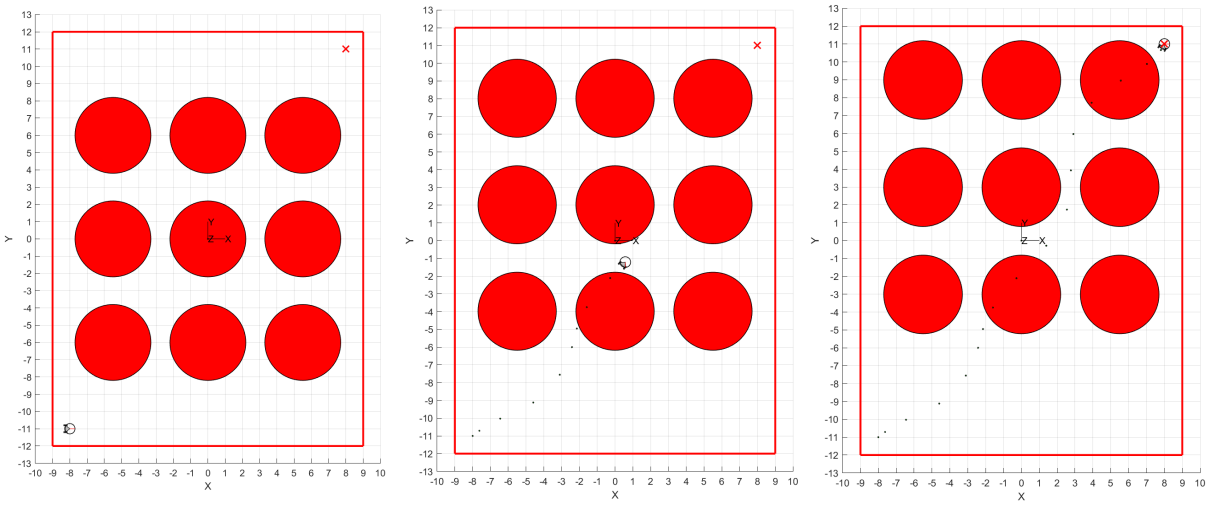


Figure 4.14 – Insane box with moving obstacles

Finally, we created a scenario to show the capabilities of this trajectory planning scheme. The scenario can be seen in Fig. 4.14 where all obstacles will be moving up and down synchronously whilst the Otbot has to move to the upper right corner as fast as possible. This problem can again be solved without providing any initial guess, however it takes some time to compute due to the nine obstacles. To note that when minimizing time only, the robot shows great use of the pivot torque, most likely to create a counter torque and speed up its turning maneuvers. When computing a new solution whilst adding a penalty on the pivot torque, we find a similar trajectory this time with almost no movement of the platform. The second solution however takes 4.93s to reach the goal position instead of the 4.69s of the first solution.

4.8 Using analytical derivatives

One major inconvenient of the trajectory optimization scheme presented here is the computation time, which can drastically increase when using high-order methods, a high number of knot points or solving for scenarios with many obstacles. This is due to the fact that the optimizer needs to compute the gradient of every equation evaluated at each collocation point. This includes the constraints (system dynamics and obstacles) and the objective function. Until now, we have been using finite differences to compute these gradients at each collocation point, which can take some time when there are a lot of points to evaluate.

The solution to speed up computation is to provide the analytical gradients to the optimizer as a function so it can compute these gradients much faster, without the need to calculate finite differences. We will now present how to obtain the gradient for the system

dynamics $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$. To obtain this gradient, we need to find the two partial derivatives:

$$\frac{\partial \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})}{\partial \mathbf{x}} \qquad \text{and} \qquad \frac{\partial \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})}{\partial \mathbf{u}}.$$

If we recall the equation of the dynamic model in first-order form as shown in Eq. (3.4.10), we can see that the partial derivatives we seek have the block structure:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial \dot{\boldsymbol{q}}}{\partial \boldsymbol{q}} & \frac{\partial \dot{\boldsymbol{q}}}{\partial \dot{\boldsymbol{q}}} \\ \frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{q}} & \frac{\partial \ddot{\boldsymbol{q}}}{\partial \dot{\boldsymbol{q}}} \end{bmatrix} = \begin{bmatrix} 0 & \boldsymbol{I}_6 \\ \frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{q}} & \frac{\partial \ddot{\boldsymbol{q}}}{\partial \dot{\boldsymbol{q}}} \end{bmatrix},$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}} = \begin{bmatrix} \frac{\partial \dot{\boldsymbol{q}}}{\partial \boldsymbol{u}} \\ \frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{u}} \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{u}} \end{bmatrix}.$$

This means we need to compute these three partial derivatives: $\frac{\partial \ddot{\boldsymbol{q}}}{\partial \dot{\boldsymbol{q}}}$, $\frac{\partial \ddot{\boldsymbol{q}}}{\partial \dot{\boldsymbol{q}}}$ and $\frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{u}}$. To avoid computing the partial derivatives of the inverse of a matrix, we will use the following trick:

$$\begin{aligned} &\boldsymbol{M}^{-1}\boldsymbol{M} = I, \\ &\frac{\partial \left(\boldsymbol{M}^{-1}\boldsymbol{M}\right)}{\partial q_i} = 0, \\ &\frac{\partial \boldsymbol{M}^{-1}}{\partial q_i} \cdot \boldsymbol{M} + \boldsymbol{M}^{-1}\frac{\partial \boldsymbol{M}}{\partial q_i} = 0, \\ &\frac{\partial \boldsymbol{M}^{-1}}{\partial q_i} = -\boldsymbol{M}^{-1}\frac{\partial \boldsymbol{M}}{\partial q_i}\boldsymbol{M}^{-1}. \end{aligned}$$

We can now compute $\frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{q}}$ for each state variable q_i in the following manner:

$$\begin{aligned} \frac{\partial \ddot{\boldsymbol{q}}}{\partial q_i} &= \frac{\partial}{\partial q_i}\left(\boldsymbol{M}^{-1}\boldsymbol{F}\right) = \\ &= \frac{\partial \boldsymbol{M}^{-1}}{\partial q_i} \cdot \boldsymbol{F} + \boldsymbol{M}^{-1}\frac{\partial \boldsymbol{F}}{\partial q_i} = \\ &= \left(-\boldsymbol{M}^{-1}\frac{\partial \boldsymbol{M}}{\partial q_i}\boldsymbol{M}^{-1}\right)\boldsymbol{F} + \boldsymbol{M}^{-1}\frac{\partial \boldsymbol{F}}{\partial q_i} = \\ &= \boldsymbol{M}^{-1}\left(\frac{\partial \boldsymbol{M}}{\partial q_i}\ddot{\boldsymbol{q}} + \frac{\partial \boldsymbol{F}}{\partial q_i}\right), \end{aligned}$$

here, $\boldsymbol{M}^{-1}$ is obtained by numerical inversion, $\ddot{\boldsymbol{q}}$ by solving $\boldsymbol{M}\ddot{\boldsymbol{q}} = \boldsymbol{F}$ and the two remaining partial derivatives are found using symbolic differentiation in Matlab.

Using a similar process as above, we also find:

$$\frac{\partial \ddot{\boldsymbol{q}}}{\partial \dot{q}_i} = \boldsymbol{M}^{-1}\frac{\partial \boldsymbol{F}}{\partial \dot{q}_i} \qquad \text{and} \qquad \frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{u}} = \boldsymbol{M}^{-1}\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{u}},$$

where the remaining partial differences are obtained by symbolic differentiation once again.

Now, since we are using the $\boldsymbol{z}$ model, we need to find the gradient for the $\boldsymbol{z}$ dynamic model $\dot{\boldsymbol{z}} = \boldsymbol{g}(\boldsymbol{z}, \boldsymbol{u})$. Using the inverse map in Eq. (4.5.4) and the derivative chain rule, it is easy to obtain the following partial derivatives:

$$\frac{\partial \boldsymbol{g}(\boldsymbol{z}, \boldsymbol{u})}{\partial \boldsymbol{z}} = \boldsymbol{L} \cdot \frac{\partial \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})}{\partial \boldsymbol{x}} \cdot \frac{\partial \boldsymbol{\Psi}_{\boldsymbol{p}}(\boldsymbol{z})}{\partial \boldsymbol{z}},$$
$$\frac{\partial \boldsymbol{g}(\boldsymbol{z}, \boldsymbol{u})}{\partial \boldsymbol{u}} = \boldsymbol{L} \cdot \frac{\partial \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})}{\partial \boldsymbol{u}}.$$

We now only need to obtain $\frac{\partial \boldsymbol{\Psi}_{\boldsymbol{p}}(\boldsymbol{z})}{\partial \boldsymbol{z}}$ which is a mostly constant sparse matrix that can be easily derived from Eq. (4.5.3), where the only variables are the partial derivatives $\frac{\partial \dot{\boldsymbol{\varphi}}}{\partial \boldsymbol{z}}$ which can be obtained with symbolic differentiation.

Providing this gradient to the optimizer allows for faster computation, as can be seen from Fig. 4.15. Here for both collocation methods and objective functions used, the computation is reduced when providing the analytical gradient. It is important to note that while these experiments were conducted on the same computer, they were run on different days, using Matlab, and with no efforts made to increase or stabilise the available processing power, which might have caused fluctuations when measuring computation time. Nevertheless, this data points to a clear improvement when using analytical gradients, and hint that further reduction of computation time is quite accessible.

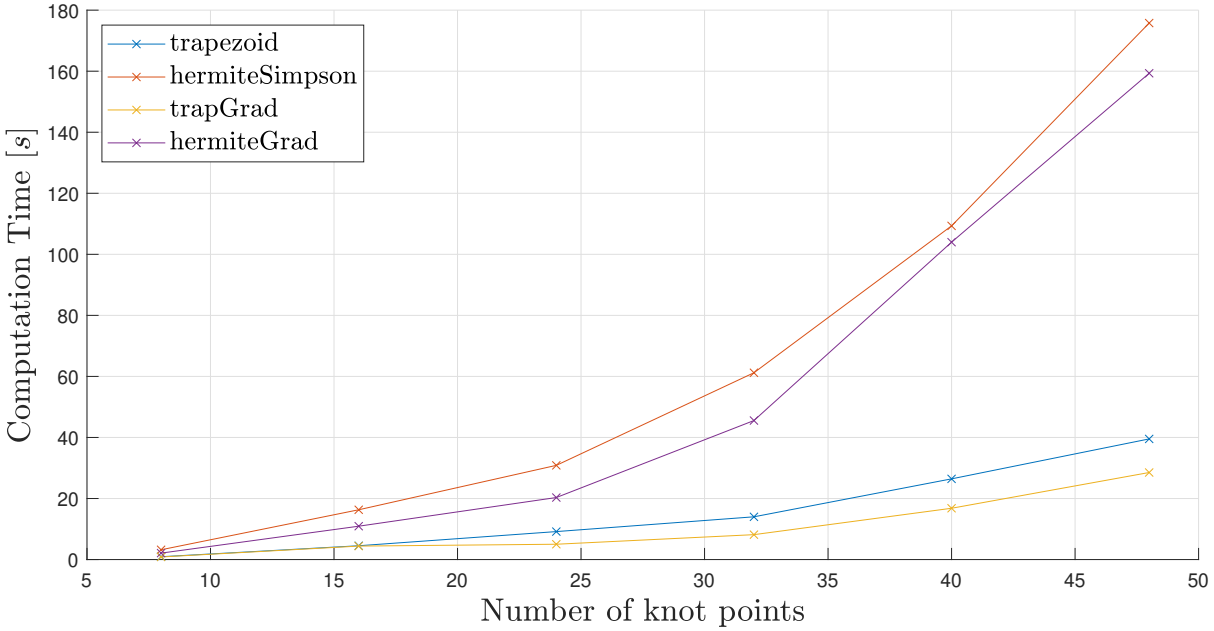


Figure 4.15 – Comparison of computation time for different collocation methods with force squared objective function for basic task

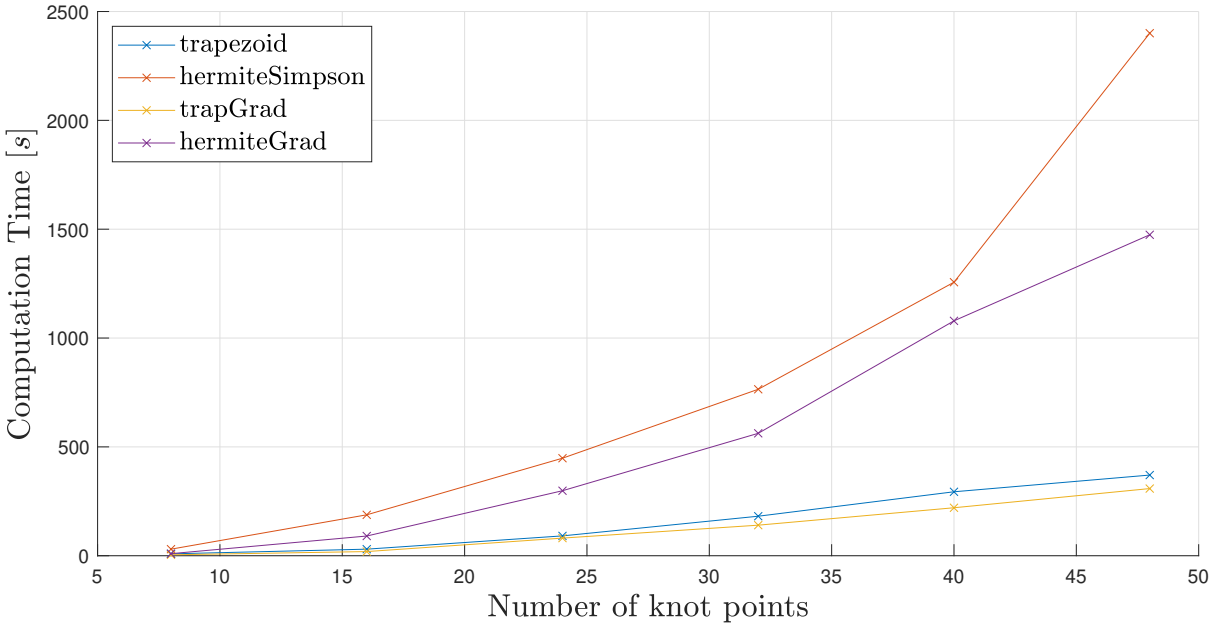


Figure 4.16 – Comparison of computation time for different collocation methods with time objective function for basic task

4.9 Implementing velocity-dependent torque bounds

Thanks to the versatility of the trajectory optimization problem formulation used here, it is possible to model almost any real physical limitations in our system. One of such limitations is the maximum torque that can be applied by a DC motor. Until now, we have been using a constant bound on this torque with a conservative value that ensured that torques required to execute a trajectory are physically feasible. This is however a simple approximation which does not properly model the behavior of a DC motor. Indeed, a constant torque bound does not take into account the velocity dependence of the torques in a DC motor. In a real DC motor, the maximum torque that can be applied depends on the angular velocity at that moment and is given by the torque-velocity curve of the motor. This curve is usually represented as seen in Fig. 4.17a, however the complete relationship between torque and velocity is better represented using Fig. 4.17b, where τ is the motor torque and $\dot{\varphi}$ the rotational speed [16]. The velocity-torque curve depends on two intrinsic parameters of the DC motor: the stall torque τ_{stall} and the no-load speed $\dot{\varphi}_{no\,load}$. They determine the slope s of the maximum and minimum torque lines in Fig. 4.17b as

$$s = -\frac{\tau_{stall}}{\dot{\varphi}_{no\,load}}.$$

We are looking for an expression of the maximum torque the actuator can apply at the output of its gearbox. For this, we use the gear box factor N which relates the speeds

and torques on the actuator and motor sides as follows:

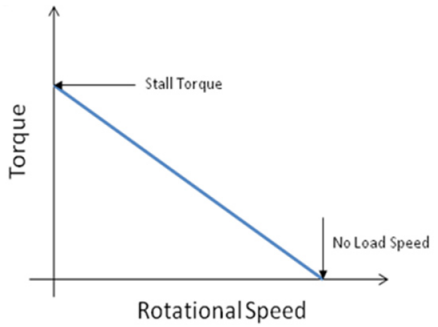
$$\dot{\varphi}^m = N \cdot \dot{\varphi},$$
$$\tau^a_{\max} = N \cdot \tau^m_{\max},$$

where ^{m} indicates the motor side and ^{a} the actuator side. $\dot{\varphi}$ is the angular velocity at the output of the gearbox. Using this ratio and the torque-velocity curve, we can write:

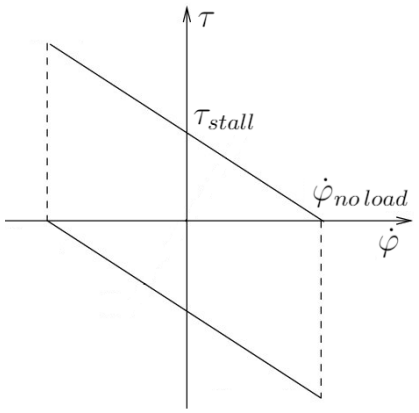
$$\tau^m_{\max}\left(\dot{\varphi}^m\right) = \tau_{\text{stall}} + s \cdot \dot{\varphi}^m,$$
$$\tau^a_{\max}(\dot{\varphi}) = N\left(\tau_{\text{stall}} + s \cdot N \cdot \dot{\varphi}\right), \tag{4.9.1}$$
$$\tau^a_{\min}(\dot{\varphi}) = N\left(-\tau_{\text{stall}} + s \cdot N \cdot \dot{\varphi}\right). \tag{4.9.2}$$

Using Eq. (4.9.1) and Eq. (4.9.2) in the path constraints, we can impose a maximum and minimum torque at each motor for any given $\dot{\varphi}$. The parameters used for the Otbot are shown in Table 4.2.

Figs. 4.18 and 4.19 show the trajectories obtained for state variables and torques using, respectively, constant and velocity-dependent torque bounds for the same basic task as in Section 4.6. We can clearly see the torque-velocity relation in the two lower plots, where the torque used decreases when the angular velocities increase. In fact, this is also a bang-bang trajectory since the robot constantly uses the maximum available torque.



(a) Torque-velocity plot



(b) Torque-velocity full plot

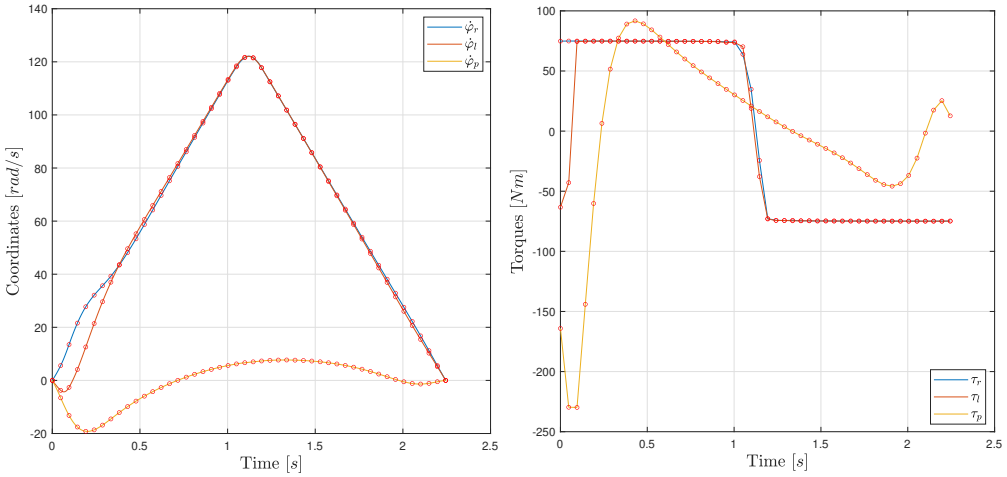
Figure 4.17 – Torque-velocity characteristics of a typical DC motor

Symbol	Meaning	Value	Unit
τ_{stall}	Stall Torque	2	[Nm]
$\dot{\varphi}_{no\,load}$	No load speed	50'000	[rpm]
N	Gear box factor for wheel motors	50	-
N_p	Gear box factor for pivot motor	150	-

Table 4.2 – Otbot Motor parameters

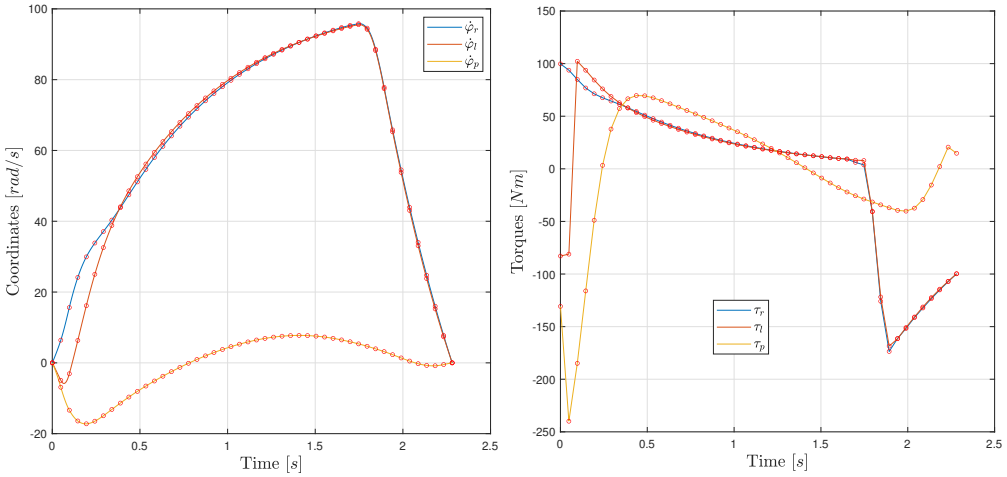
This is especially visible when the motors change direction at about 1.75s, and the torque exerted rapidly changes from nearly 0 to almost $-2 \cdot N \cdot \tau_{stall}$, the maximum available negative torque at maximum positive velocity. Notice also the new asymmetry of the torque profile: since more torque is available when switching direction, the robot needs less time to brake before arriving. In contrast, the trajectory with symmetric constant bounds switches motor direction in the middle of the trajectory since it applies maximum torque as long as possible before applying the same opposite maximal torque to brake.

Applying these bounds only barely increases the time necessary to execute the trajectory, since the slower acceleration is compensated by a shorter and harder brake. Additionally, it is a much more realistic behavior of the motors. Adding these constraint, similarly to adding obstacles, does slightly increase computation time since more values need to be calculated at each collocation point, however the difference for this scenario is not significant.



(a) $\dot{\boldsymbol{\varphi}}$-state variables (b) Torque profile

Figure 4.18 – basic task with constant torque bounds



(a) $\dot{\boldsymbol{\varphi}}$-state variables (b) Torque profile

Figure 4.19 – basic task with velocity-dependent torque bounds

Chapter 5

Trajectory Control

Once a trajectory as been planned by the optimizer, we need a controller to allow the robot to track the desired trajectory since simply using the control from the optimizer will not suffice due to the inaccuracies of the discretization. For this purpose, we designed a globally stable computed-torque controller capable of tracking any desired trajectory from any starting position. This controller will also allow the robot to compensate unmodelled force perturbations.

5.1 A computed-torque controller

A computed-torque controller can only control as many variables as the number of action coordinates. Since there are only three control coordinates in the Otbot (two wheels and one pivot), we need to base this controller on an equation of motion which uses only three state variables. Fortunately, we have already computed the equation of motion in task-space coordinates $\boldsymbol{p} = (x, y, \alpha)$ in Eq. (3.4.8), which we recall for convenience:

$$\bar{\boldsymbol{M}}\ddot{\boldsymbol{p}} + \bar{\boldsymbol{C}}\dot{\boldsymbol{p}} = \boldsymbol{u}. \tag{5.1.1}$$

This equation is ideal since it will let us control the **p**-state variables, which are those typically used to specify a task or mission for the robot. We can convert the system in Eq. (5.1.1) into a linear double integrator by using this feedback law:

$$\boldsymbol{u} = \bar{\boldsymbol{M}}\boldsymbol{u}' + \bar{\boldsymbol{C}}\dot{\boldsymbol{p}},$$

which, if we substitute back into Eq. (5.1.1) cancels all linearities and gives the very simple linear system:

$$\ddot{\boldsymbol{p}} = \boldsymbol{u}'. \tag{5.1.2}$$

With this feedback law, we can now apply linear control techniques to the Otbot. Since we wish to have the Otbot match a desired trajectory $\boldsymbol{p_d}(t)$ from any given position, we use this second control law on top of the previous one

$$\boldsymbol{u}' = \ddot{\boldsymbol{p}}_{\boldsymbol{d}}(t) + \boldsymbol{K_p}(\boldsymbol{p_d}(t) - \boldsymbol{p}(t)) + \boldsymbol{K_v}(\dot{\boldsymbol{p}}_{\boldsymbol{d}}(t) - \dot{\boldsymbol{p}}(t)), \tag{5.1.3}$$

where $\boldsymbol{K_p}$ and $\boldsymbol{K_v}$ are positive-definite gain matrices.

To wrap up, our system in Eq. (3.4.8) can be made to converge to our desired trajectory $\boldsymbol{p_d}(t)$ using the computed-torque control law:

$$\boldsymbol{u} = \bar{\boldsymbol{M}}(\ddot{\boldsymbol{p}}_{\boldsymbol{d}}(t) + \boldsymbol{K_p}(\boldsymbol{p_d}(t) - \boldsymbol{p}(t)) + \boldsymbol{K_v}(\dot{\boldsymbol{p}}_{\boldsymbol{d}}(t) - \dot{\boldsymbol{p}}(t))) + \bar{\boldsymbol{C}}\dot{\boldsymbol{p}}.$$

Videos showing the performances of the controller in simulation are available in this playlist, with plots showing the error along the trajectory in the Appendix B. Videos and plots showing simulations for the same desired trajectories but using the control provided by the optimizer are also available.

5.2 Global stability of the controller

The main reason for using a computed-torque controller is that it makes the system globally stable. This means that the system will converge asymptotically to $\boldsymbol{p}_d(t)$ irrespective of its initial state $\boldsymbol{p}(0)$. We next demonstrate this point.

If we substitute Eq. (5.1.3) into our linear system in Eq. (5.1.2) we get

$$0 = \ddot{\boldsymbol{p}}_{\boldsymbol{d}}(t) - \ddot{\boldsymbol{p}}(t) + \boldsymbol{K_p}(\boldsymbol{p_d}(t) - \boldsymbol{p}(t)) + \boldsymbol{K_v}(\dot{\boldsymbol{p}}_{\boldsymbol{d}}(t) - \dot{\boldsymbol{p}}(t))$$

and by noting that $\boldsymbol{\varepsilon}(t) = \boldsymbol{p_d}(t) - \boldsymbol{p}(t)$ is the configuration error, we obtain

$$0 = \ddot{\boldsymbol{\varepsilon}}(t) + \boldsymbol{K_p}\boldsymbol{\varepsilon}(t) + \boldsymbol{K_v}\dot{\boldsymbol{\varepsilon}}(t),$$

which is the ordinary differential equation governing the time evolution of $\boldsymbol{\varepsilon}(t)$. For such an equation, it is well-known that

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

if $\boldsymbol{K_p}$ and $\boldsymbol{K_v}$ are positive-definite [13], so we clearly have

$$\lim_{t\to\infty} \boldsymbol{p}(t) = \boldsymbol{p_d}(t)$$

when the gain matrices are chosen as we assumed.

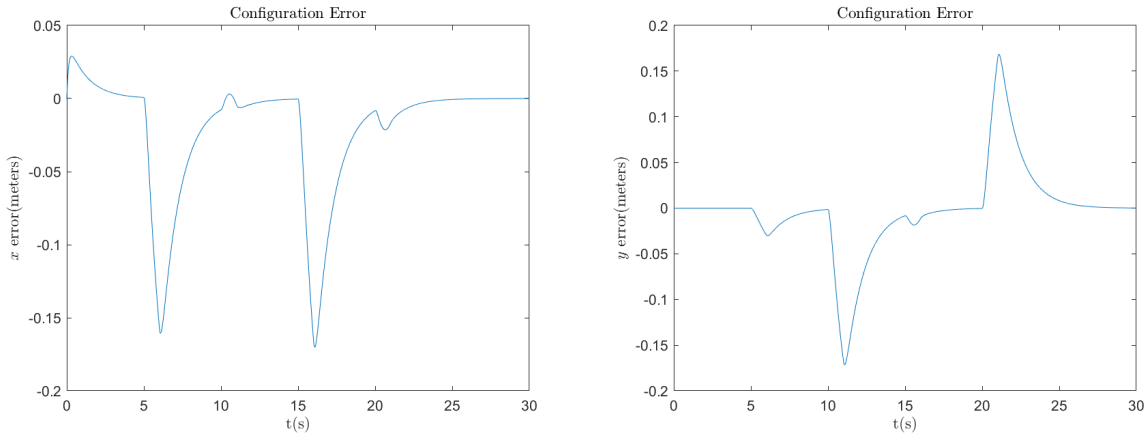
5.3 Tuning of the control law

To achieve a proper control, we must tune our controller by setting adequate values for $\boldsymbol{K_p}$ and $\boldsymbol{K_v}$. To do this, we use the full state feedback or pole placement method which lets us place the closed-loop poles of our plant in a pre-determined location in the s-plane. This is necessary since the poles correspond to the eigenvalues of our control system, which define the characteristics of the system response. To actually find $\boldsymbol{K_p}$ and $\boldsymbol{K_v}$ we apply the method in [17], which ensures that the sensitivity of the assigned poles to perturbations in the system and gain matrices is minimized.

According to linear control theory, in order to get a stable closed-loop system we need all poles with a negative real part. A precise placement of the poles takes into account how fast the robot should converge to step or ramp inputs according to given specifications. In our case, we placed the poles through trial and error in order to find reasonable values which gave us both a stable and fast response close to that of an over-damped second-order system.

5.4 Disturbance rejection tests

By design, the obtained controller is capable of compensating for disturbances along the trajectory, which we will show here. We have modeled disturbances along a circular trajectory as forces pushing the robot to the side. They are visible in this video as red arrows. We can clearly see here that the robot converges back to the desired circular trajectory as soon as possible after each disturbance. We can also observe the efficacy of the controller in Fig. 5.1 where each spike in the error corresponds to a disturbance which is immediately followed by a compensation from the controller.



(a) Error in X

(b) Error in Y

Figure 5.1 – Configuration errors for the circle follow trajectory with disturbances

Chapter 6

Conclusion

Throughout this project we have established a method for optimal trajectory planning as well as a controller for an omnidirectional tire-wheeled robot. We first developed a kinematic model by defining the three velocity constraints which impose no sliding, as well as one integral holonomic constraint. We then constructed a complete dynamical model which, compared with previous works [7, 8, 9], has been generalized for an arbitrary position of the centers of mass of the different parts of the robot. We attempted to use this model with an obvious parametrization to solve for optimal trajectories. However, due to the dependencies in state coordinates several errors, mainly kinematic, appeared in the obtained trajectories. These errors were avoided by using a new model with independent state coordinates only, which eliminates kinematic errors by construction.

Using this new model, we were able to compute optimal trajectories for a number of different scenarios. We compared two different collocation methods: trapezoidal and Hermite-Simpson, with a varying number of knot points. This showed the typical trade-off of trajectory optimization: using more accurate methods leads to a reduction in dynamical error up to a point, at the cost of a high computation time. This means the ideal settings to solve a trajectory optimization problem depend on both the accuracy desired for the obtained trajectory, and the available time for computation. Additionally, we also compared several cost functions: minimizing for total trajectory time, for the torque-squared applied during the trajectory, for the torque rate, or even a combination of these. Different cost functions force different behaviors in the robot. When minimizing for time, we obtained near optimal bang-bang trajectories where some of the motor torques was either maximal or minimal; when minimizing torque-squared we instead obtained smoother trajectories using smaller motor actions; and when minimizing the torque rate, finally, the resulting controls turned to be very smooth, which is useful to ease their tracking.

Moreover, the problem formulation typically assumed in direct collocation methods was shown to be extremely versatile, handling constant or state dependant motor torque bounds, as well as any other bounds on the states and actions of the model. In fact, we implemented a simple way to model a wide range of desired scenarios, constraints or cost for the Otbot. Obstacles, whether static or moving, can also be easily modelled to create relatively complex environments. More importantly, near-optimal trajectories

were successfully found for every tested configuration in reasonable computation times. Further improvement was also made by providing analytical gradients to the solver, which speeds up the calculations as we have shown. Finally, we designed a globally stable robust controller for the Otbot capable of tracking and following any desired trajectory in its task space.

The results provided in this thesis show the feasibility and advantages of controlling and planning trajectories for a tire-wheeled mobile robot with an omnidirectional platform. Such a robot could be constructed to service warehouses or factories, where it could carry heavy loads without needing complex manoeuvres. In fact, if augmented with a motion planner, our trajectory planner and controller could work autonomously to find, optimize, and follow near-optimal trajectories. While the motion planner would endow the robot with the ability to search the whole motion space for a feasible trajectory, the trajectory optimizer could be used to optimize the planned trajectory according to a cost function of interest. However, in order to be used as an on-board planner, the computation time for our trajectory optimization scheme should be reduced to provide feasible, near-optimal trajectories in sufficiently short times. This could probably be achieved by re-implementing our optimizer in some compilable language like C or C++, so as to be able to run a compiled software rather than interpreted code as the one we have now in Matlab. A further improvement would be to design a linear-quadratic regulator controller based on the $\boldsymbol{z}$ model of the Otbot. This controller would fully control the Otbot so that every state variable follows the planned optimal trajectory precisely.

An omnidirectional tire-wheeled robot as defined here, equipped with a motion planner, an accurate trajectory optimizer, and a robust controller could become an invaluable autonomous help for moving heavy loads in warehouses or factories in the following years.

Bibliography

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] G. Campion and W. Chung, "Wheeled robots," in *Springer Handbook of Robotics*. Springer Berlin Heidelberg, 2008, pp. 391–410.

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

[4] J. Agulló, S. Cardona, and J. Vivancos, "Kinematics of vehicles with directional sliding wheels," *Mechanism and Machine Theory*, vol. 22, no. 4, pp. 295–301, jan 1987.

[5] ——, "Dynamics of vehicles with directionally sliding wheels," *Mechanism and Machine Theory*, vol. 24, no. 1, pp. 53–60, jan 1989.

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

[7] 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.

[8] 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.

[9] 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.

[10] J. T. Betts, "Survey of numerical methods for trajectory optimization," *Journal of Guidance, Control, and Dynamics*, vol. 21, no. 2, pp. 193–207, 1998.

[11] ——, *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Society for Industrial and Applied Mathematics, 2010.

[12] M. Kelly, "An introduction to trajectory optimization: How to do your own direct collocation," *SIAM Review*, vol. 59, no. 4, pp. 849–904, 2017.

[13] R. M. Murray, S. S. Li, Zexiang, and Sastry, *A mathematical introduction to robotic manipulation*. CRC press, 1994.

[14] D. Liberzon, *Calculus of variations and optimal control theory: a concise introduction*. Princeton university press, 2011.

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

[16] T. Kalmar-Nagy, P. Ganguly, and R. D'Andrea, "Real-time trajectory generation for omnidirectional vehicles," in *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, vol. 1, 2002, pp. 286–291 vol.1.

[17] J. Kautsky, N. K. Nichols, and P. V. Dooren, "Robust pole assignment in linear state feedback," *International Journal of Control*, vol. 41, no. 5, pp. 1129–1155, 1985.

Appendix A

Expressions used to define Otbot model

In this appendix, the expressions for the translational and rotational kinetic energy of each body of the Otbot can be found. They are expressed using the state variables and robot parameters as defined in Chapter 2 and 3.

A.1 Translational kinetic energy for each body

A.1.1 T_t chassis

$$
\begin{aligned}
T_{t,\,chassis} = {} & \frac{m_b\,\dot{\alpha}^2\,{x_G}^2}{2} + \frac{m_b\,\dot{\alpha}^2\,{y_G}^2}{2} - m_b\,\dot{\alpha}\,\dot{\varphi}_p\,{x_G}^2 - m_b\,\dot{\alpha}\,\dot{\varphi}_p\,{y_G}^2 \\
& - m_b \sin(\alpha - \varphi_p)\ \dot{\alpha}\,x_G\,\dot{x} + m_b \cos(\alpha - \varphi_p)\ \dot{\alpha}\,x_G\,\dot{y} \\
& - m_b \cos(\alpha - \varphi_p)\ \dot{\alpha}\,\dot{x}\,y_G - m_b \sin(\alpha - \varphi_p)\ \dot{\alpha}\,y_G\,\dot{y} + \\
& \frac{m_b\,\dot{\varphi}_p^2\,{x_G}^2}{2} + \frac{m_b\,\dot{\varphi}_p^2\,{y_G}^2}{2} + m_b \sin(\alpha - \varphi_p)\ \dot{\varphi}_p\,x_G\,\dot{x} \\
& - m_b \cos(\alpha - \varphi_p)\ \dot{\varphi}_p\,x_G\,\dot{y} + m_b \cos(\alpha - \varphi_p)\ \dot{\varphi}_p\,\dot{x}\,y_G \\
& + m_b \sin(\alpha - \varphi_p)\ \dot{\varphi}_p\,y_G\,\dot{y} + \frac{m_b\,\dot{x}^2}{2} + \frac{m_b\,\dot{y}^2}{2}
\end{aligned}
$$

A.1.2 T_t right wheel

$$
\begin{aligned}
T_{t,\,right\ wheel} = {} & \frac{m_w\,\dot{\alpha}^2\,{l_1}^2}{2} + \frac{m_w\,\dot{\alpha}^2\,{l_2}^2}{2} - m_w\,\dot{\alpha}\,{l_1}^2\,\dot{\varphi}_p + m_w \sin(\alpha - \varphi_p)\ \dot{\alpha}\,l_1\,\dot{x} \\
& - m_w \cos(\alpha - \varphi_p)\ \dot{\alpha}\,l_1\,\dot{y} - m_w\,\dot{\alpha}\,{l_2}^2\,\dot{\varphi}_p + m_w \cos(\alpha - \varphi_p)\ \dot{\alpha}\,l_2\,\dot{x} \\
& + m_w \sin(\alpha - \varphi_p)\ \dot{\alpha}\,l_2\,\dot{y} + \frac{m_w\,{l_1}^2\,\dot{\varphi}_p^2}{2} - m_w \sin(\alpha - \varphi_p)\ l_1\,\dot{\varphi}_p\,\dot{x} \\
& + m_w \cos(\alpha - \varphi_p)\ l_1\,\dot{\varphi}_p\,\dot{y} + \frac{m_w\,{l_2}^2\,\dot{\varphi}_p^2}{2} - m_w \cos(\alpha - \varphi_p)\ l_2\,\dot{\varphi}_p\,\dot{x} \\
& - m_w \sin(\alpha - \varphi_p)\ l_2\,\dot{\varphi}_p\,\dot{y} + \frac{m_w\,\dot{x}^2}{2} + \frac{m_w\,\dot{y}^2}{2}
\end{aligned}
$$

A.1.3 T_t left wheel

$$\begin{aligned} T_{t,\,left\ wheel} = {} & \frac{m_w\,\dot{\alpha}^2\,{l_1}^2}{2} + \frac{m_w\,\dot{\alpha}^2\,{l_2}^2}{2} - m_w\,\dot{\alpha}\,{l_1}^2\,\dot{\varphi}_p + m_w \sin(\alpha-\varphi_p)\ \dot{\alpha}\,l_1\,\dot{x} \\ & - m_w \cos(\alpha-\varphi_p)\ \dot{\alpha}\,l_1\,\dot{y} - m_w\,\dot{\alpha}\,{l_2}^2\,\dot{\varphi}_p - m_w \cos(\alpha-\varphi_p)\ \dot{\alpha}\,l_2\,\dot{x} \\ & - m_w \sin(\alpha-\varphi_p)\ \dot{\alpha}\,l_2\,\dot{y} + \frac{m_w\,{l_1}^2\,\dot{\varphi}_p^2}{2} - m_w \sin(\alpha-\varphi_p)\ l_1\,\dot{\varphi}_p\,\dot{x} \\ & + m_w \cos(\alpha-\varphi_p)\ l_1\,\dot{\varphi}_p\,\dot{y} + \frac{m_w\,{l_2}^2\,\dot{\varphi}_p^2}{2} + m_w \cos(\alpha-\varphi_p)\ l_2\,\dot{\varphi}_p\,\dot{x} \\ & + m_w \sin(\alpha-\varphi_p)\ l_2\,\dot{\varphi}_p\,\dot{y} + \frac{m_w\,\dot{x}^2}{2} + \frac{m_w\,\dot{y}^2}{2} \end{aligned}$$

A.1.4 T_t platform

$$\begin{aligned} T_{t,\,platform} = {} & \frac{m_p\,\dot{\alpha}^2\,{x_F}^2}{2} + \frac{m_p\,\dot{\alpha}^2\,{y_F}^2}{2} - m_p \sin(\alpha)\ \dot{\alpha}\,x_F\,\dot{x} + m_p \cos(\alpha)\ \dot{\alpha}\,x_F\,\dot{y} \\ & - m_p \cos(\alpha)\ \dot{\alpha}\,\dot{x}\,y_F - m_p \sin(\alpha)\ \dot{\alpha}\,y_F\,\dot{y} + \frac{m_p\,\dot{x}^2}{2} + \frac{m_p\,\dot{y}^2}{2} \end{aligned}$$

A.2 Rotational kinetic energy for each body

A.2.1 T_r chassis body

$$\frac{I_b\,(\dot{\alpha}-\dot{\varphi}_p)^2}{2}$$

A.2.2 T_r rigth wheel

$$\frac{I_a\,\dot{\varphi}_r^2}{2} + \frac{I_t\,(\dot{\alpha}-\dot{\varphi}_p)^2}{2}$$

A.2.3 T_r left wheel

$$\frac{I_a\,\dot{\varphi}_l^2}{2} + \frac{I_t\,(\dot{\alpha}-\dot{\varphi}_p)^2}{2}$$

A.2.4 T_r platform

$$\frac{I_p\,\dot{\alpha}^2}{2}$$

A.3 Total kinetic energy of the Otbot

$$
\begin{aligned}
T = {} & \frac{I_p\,\dot{\alpha}^2}{2} + \frac{I_a\,\dot{\varphi}_l^2}{2} + \frac{I_a\,\dot{\varphi}_r^2}{2} + \frac{I_b\,(\dot{\alpha}-\dot{\varphi}_p)^2}{2} + I_t\,(\dot{\alpha}-\dot{\varphi}_p)^2 + \frac{m_b\,\dot{x}^2}{2} + \frac{m_p\,\dot{x}^2}{2} + m_w\,\dot{x}^2 \\
& + \frac{m_b\,\dot{y}^2}{2} + \frac{m_p\,\dot{y}^2}{2} + m_w\,\dot{y}^2 + \dot{\alpha}^2\,{l_1}^2\,m_w + \dot{\alpha}^2\,{l_2}^2\,m_w + \frac{\dot{\alpha}^2\,m_b\,{x_G}^2}{2} + \frac{\dot{\alpha}^2\,m_p\,{x_F}^2}{2} + \\
& \frac{\dot{\alpha}^2\,m_b\,{y_G}^2}{2} + \frac{\dot{\alpha}^2\,m_p\,{y_F}^2}{2} + {l_1}^2\,m_w\,\dot{\varphi}_p^2 + {l_2}^2\,m_w\,\dot{\varphi}_p^2 + \frac{m_b\,\dot{\varphi}_p^2\,{x_G}^2}{2} + \frac{m_b\,\dot{\varphi}_p^2\,{y_G}^2}{2} \\
& - 2\,\dot{\alpha}\,{l_1}^2\,m_w\,\dot{\varphi}_p - 2\,\dot{\alpha}\,{l_2}^2\,m_w\,\dot{\varphi}_p - \dot{\alpha}\,m_b\,\dot{\varphi}_p\,{x_G}^2 - \dot{\alpha}\,m_b\,\dot{\varphi}_p\,{y_G}^2 - 2\,\dot{\alpha}\,l_1\,m_w\,\dot{y}\cos(\alpha-\varphi_p) \\
& + \dot{\alpha}\,m_b\,x_G\,\dot{y}\cos(\alpha-\varphi_p) - \dot{\alpha}\,m_b\,\dot{x}\,y_G\cos(\alpha-\varphi_p) + 2\,\dot{\alpha}\,l_1\,m_w\,\dot{x}\sin(\alpha-\varphi_p) \\
& + 2\,l_1\,m_w\,\dot{\varphi}_p\,\dot{y}\cos(\alpha-\varphi_p) - \dot{\alpha}\,m_b\,x_G\,\dot{x}\sin(\alpha-\varphi_p) - \dot{\alpha}\,m_b\,y_G\,\dot{y}\sin(\alpha-\varphi_p) \\
& - m_b\,\dot{\varphi}_p\,x_G\,\dot{y}\cos(\alpha-\varphi_p) + m_b\,\dot{\varphi}_p\,\dot{x}\,y_G\cos(\alpha-\varphi_p) - 2\,l_1\,m_w\,\dot{\varphi}_p\,\dot{x}\sin(\alpha-\varphi_p) \\
& + m_b\,\dot{\varphi}_p\,x_G\,\dot{x}\sin(\alpha-\varphi_p) + m_b\,\dot{\varphi}_p\,y_G\,\dot{y}\sin(\alpha-\varphi_p) + \dot{\alpha}\,m_p\,x_F\,\dot{y}\cos(\alpha) \\
& - \dot{\alpha}\,m_p\,\dot{x}\,y_F\cos(\alpha) - \dot{\alpha}\,m_p\,x_F\,\dot{x}\sin(\alpha) - \dot{\alpha}\,m_p\,y_F\,\dot{y}\sin(\alpha)
\end{aligned}
$$

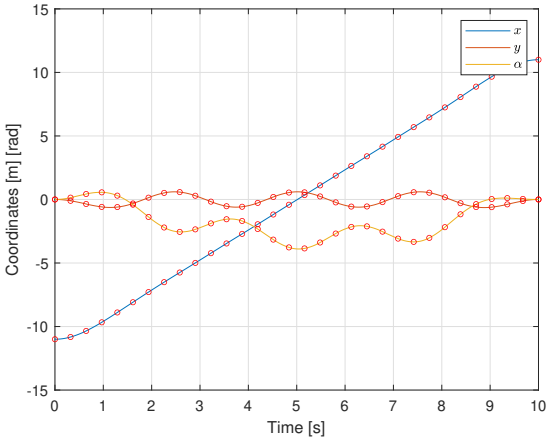
Appendix B

Simulation Plots

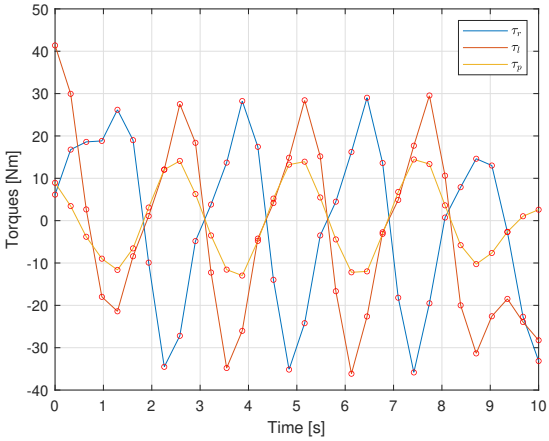
In this appendix are the $\boldsymbol{p}$-state, torque and error profiles corresponding to the trajectories shown in videos. First are the plots corresponding to obstacle avoidance trajectories. They are ordered as presented in the report and as available in this playlist.

The second section of this appendix corresponds to trajectory following using either the computed-torque controller or the control given by the optimizer. When comparing the tracking error, it is easy to see why a controller is necessary to correctly follow optimal trajectories. A few examples are available in this playlist. The code used to compute everything in this report is available on this Github.

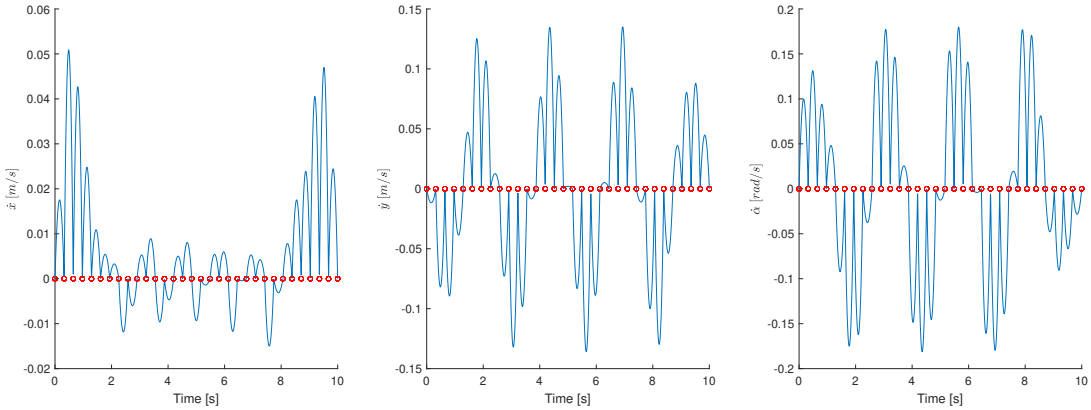
B.1 Obstacle avoidance



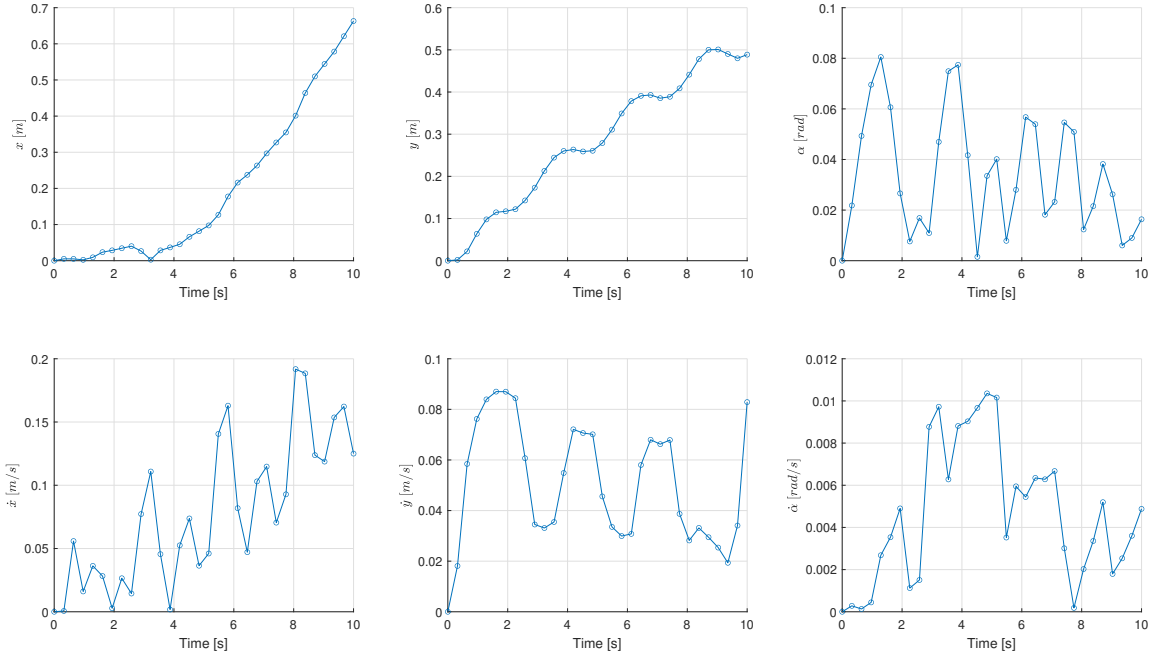
(a) $\boldsymbol{p}$-state variables



(b) torques profile

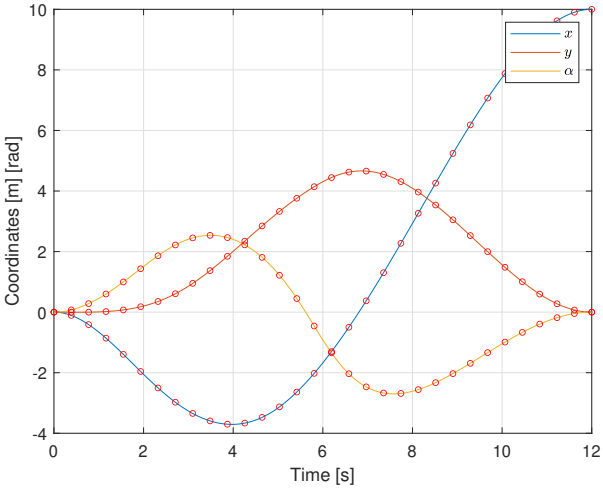


(c) Dynamical error

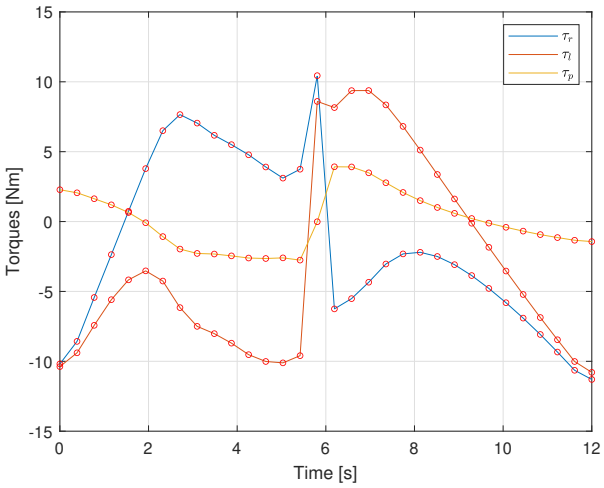


(d) Trajectory error

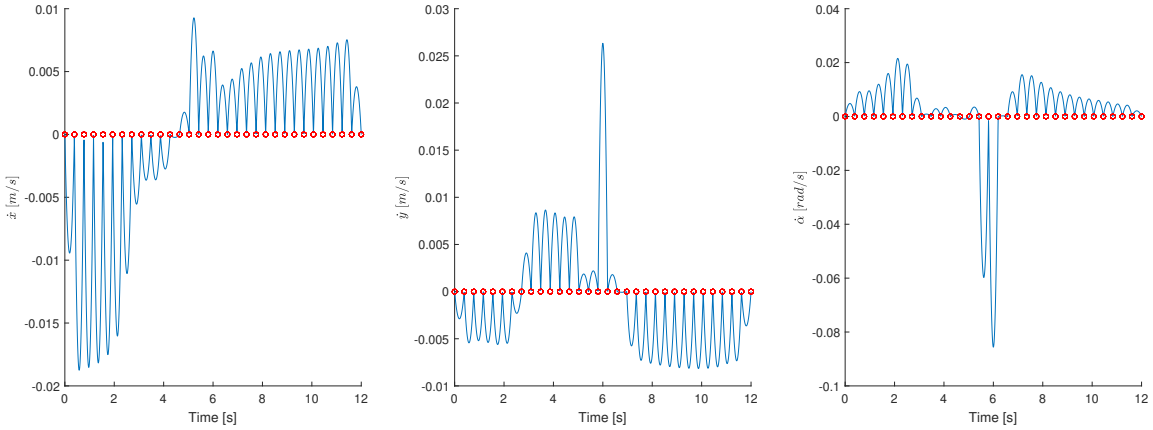
Figure B.1 – Plots corresponding to the crowded corridor trajectory seen here



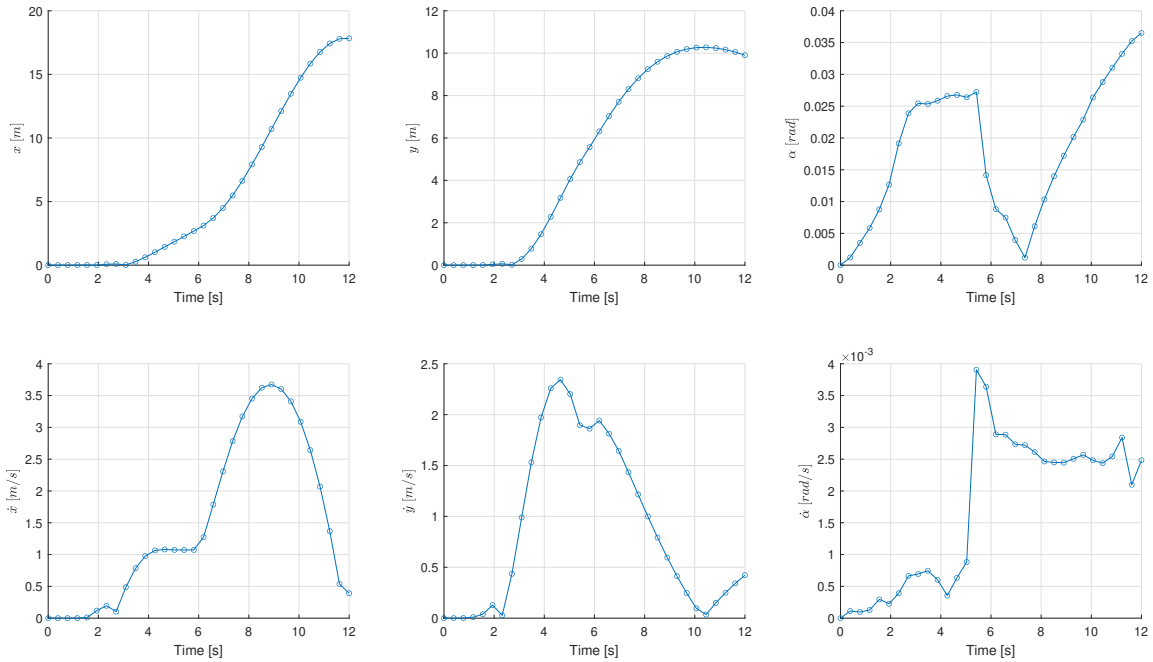
(a) $\boldsymbol{p}$-state variables



(b) torques profile

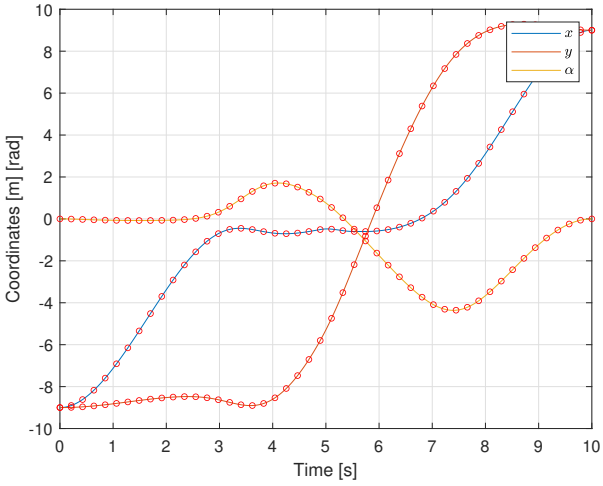


(c) Dynamical error

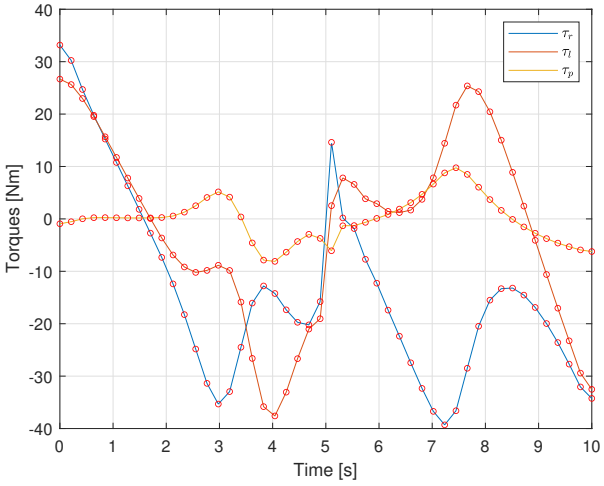


(d) Trajectory error

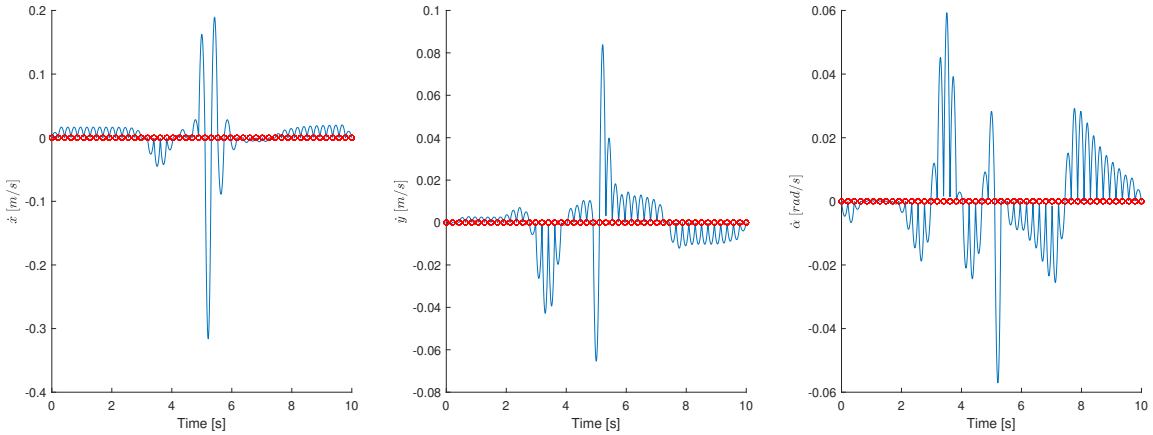
Figure B.2 – Plots corresponding to the bug trap trajectory seen here



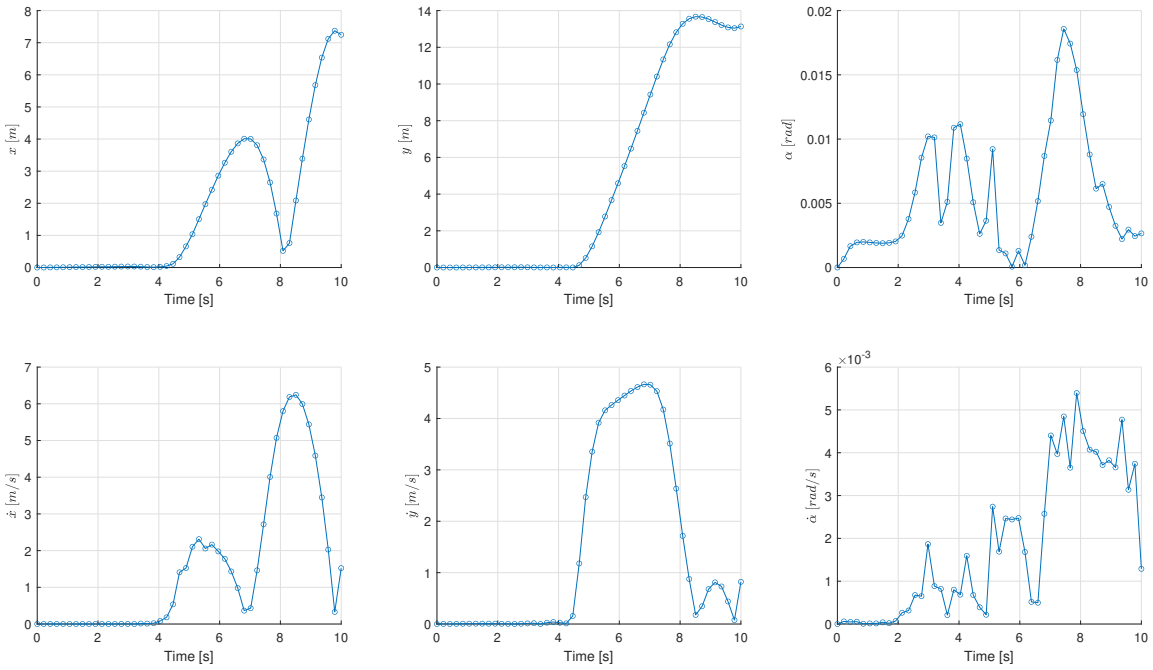
(a) $\boldsymbol{p}$-state variables



(b) torques profile

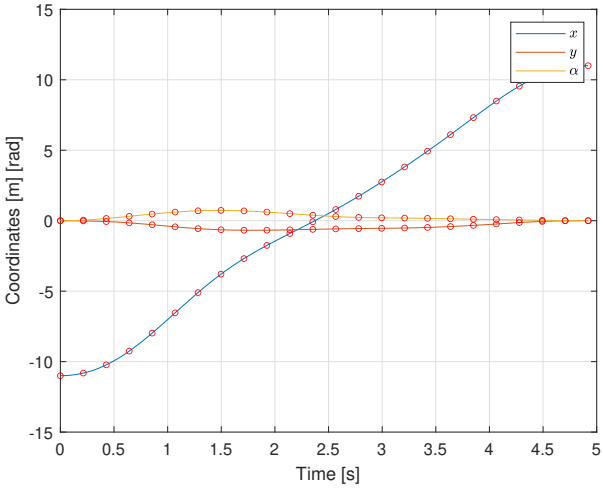


(c) Dynamical error

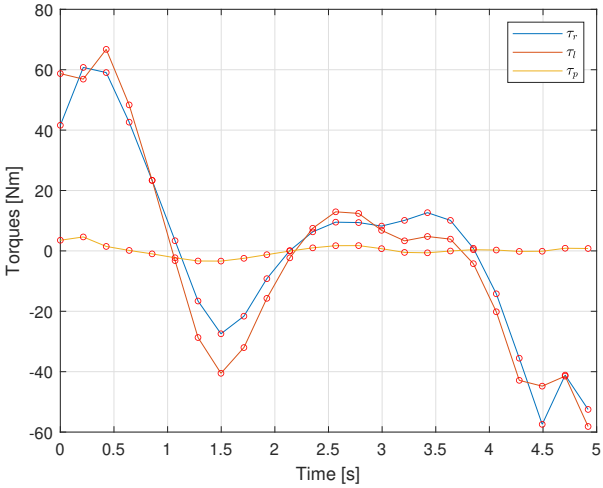


(d) Trajectory error

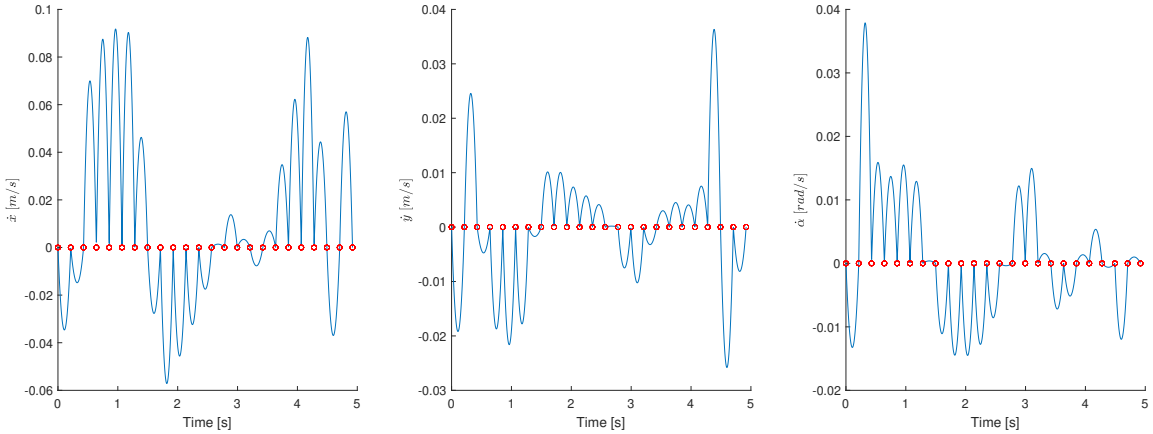
Figure B.3 – Plots corresponding to the forced deviation trajectory seen here



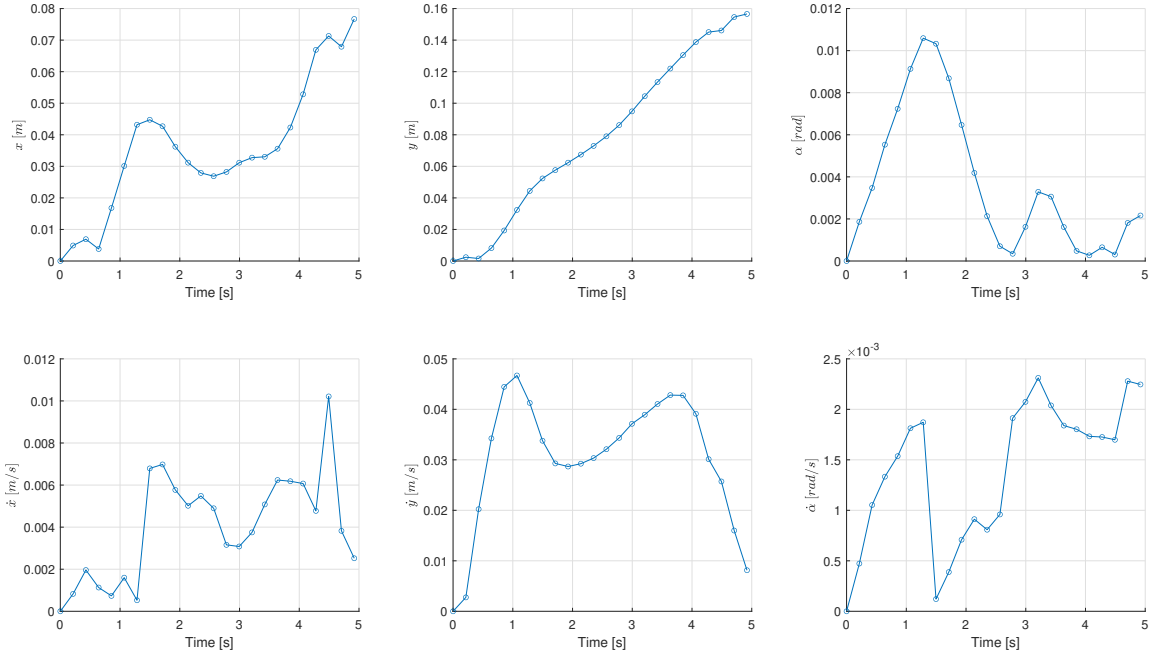
(a) $\boldsymbol{p}$-state variables



(b) torques profile

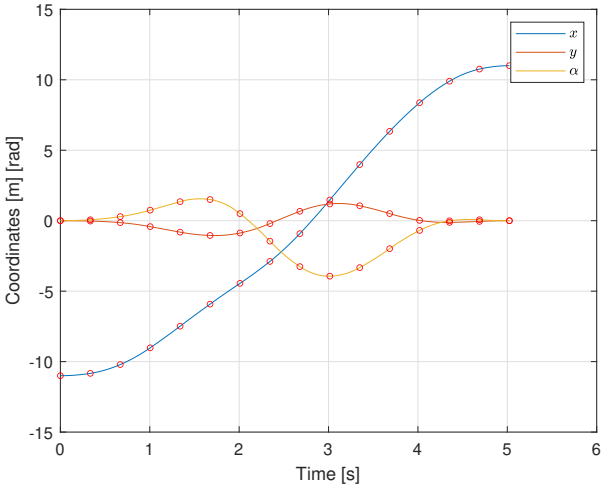


(c) Dynamical error

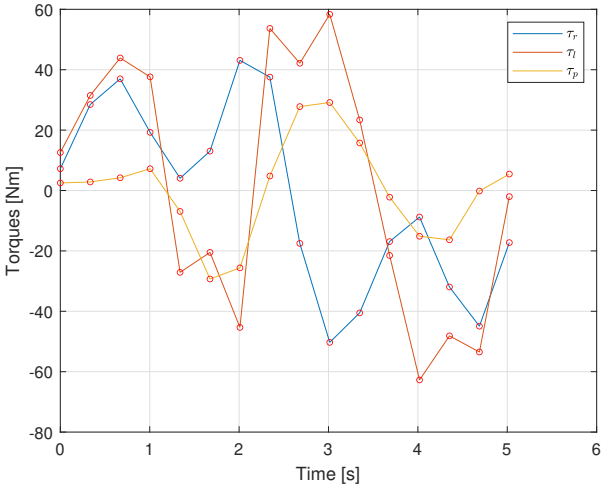


(d) Trajectory Error

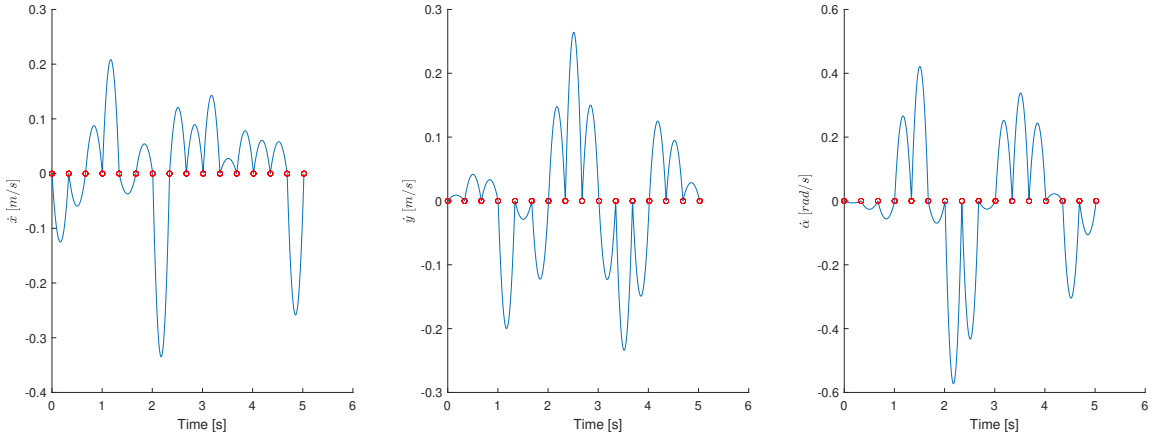
Figure B.4 – Plots corresponding to the dynamic crowded corridor (3 obstacles) trajectory seen here



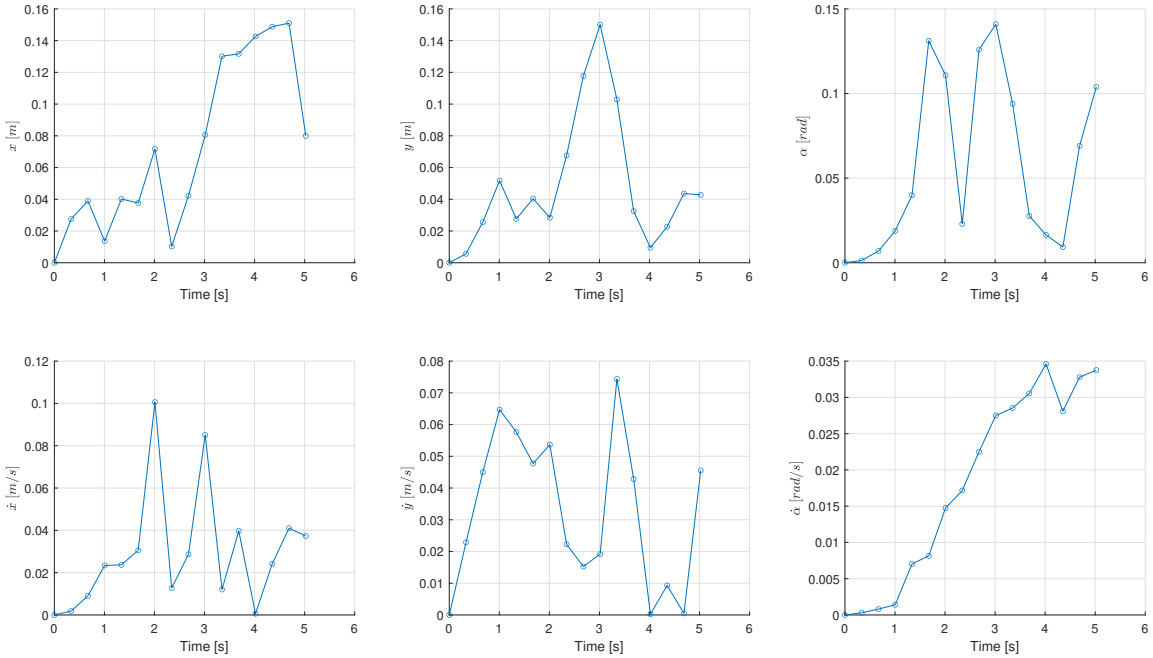
(a) $\boldsymbol{p}$-state variables



(b) torques profile

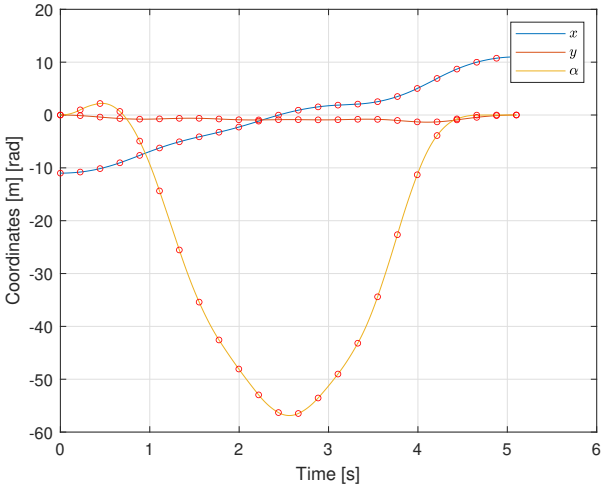


(c) Dynamical error

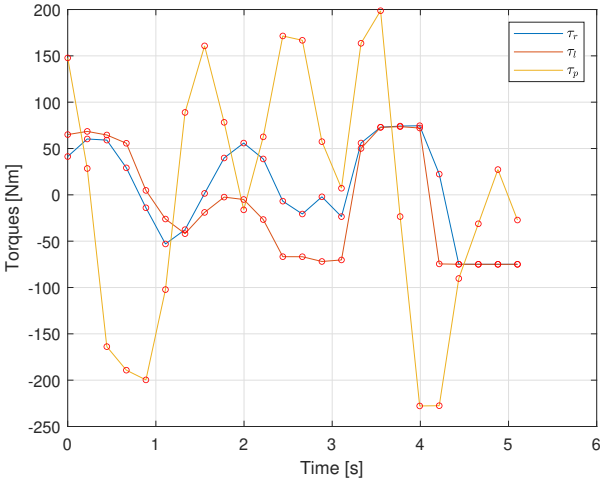


(d) Trajectory error

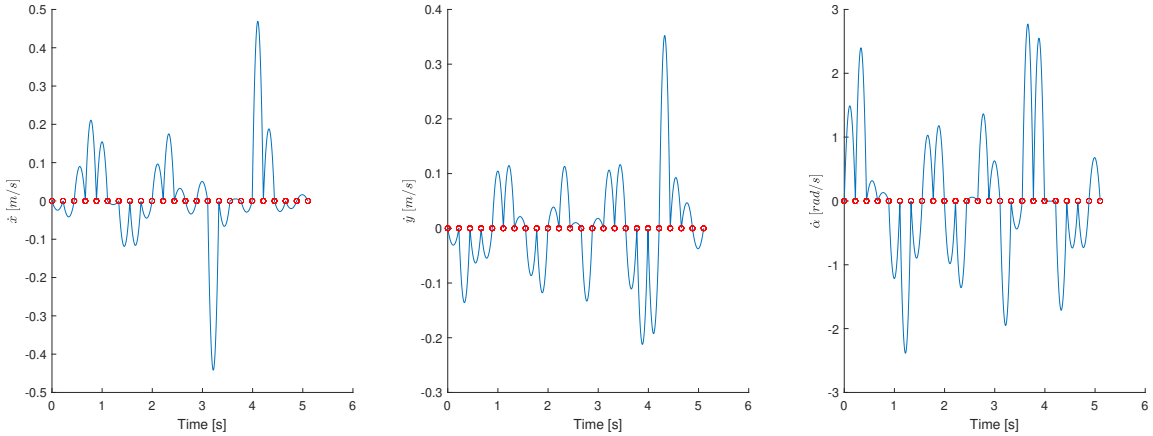
Figure B.5 – Plots corresponding to the dynamic crowded corridor (5 slow obstacles) trajectory seen here



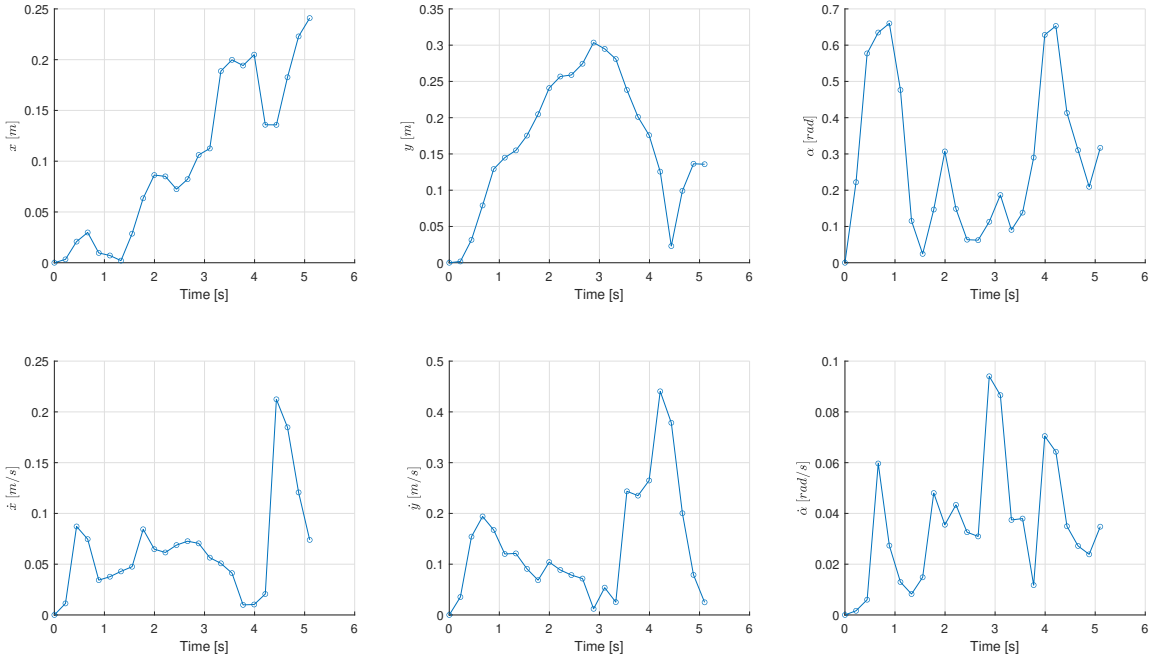
(a) $\boldsymbol{p}$-state variables



(b) torques profile

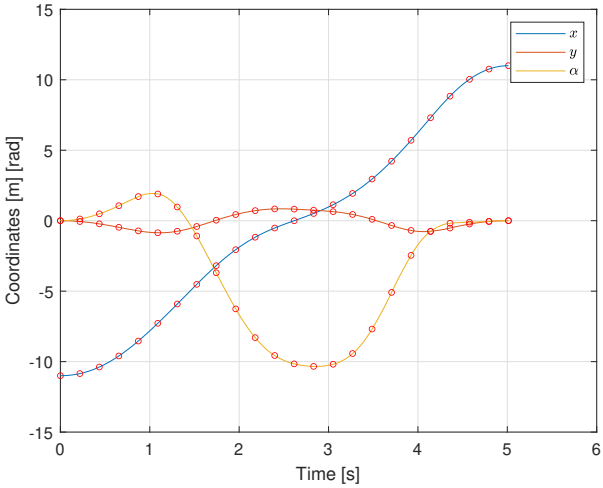


(c) Dynamical error

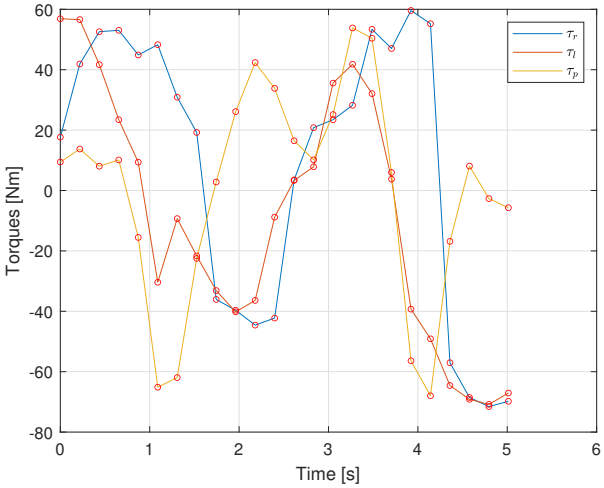


(d) Trajectory error

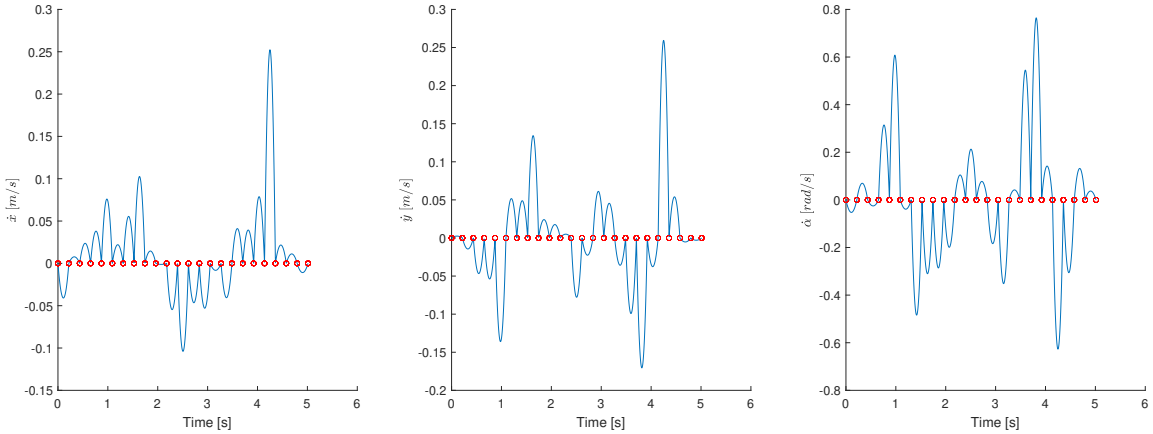
Figure B.6 – Plots corresponding to the dynamic crowded corridor (5 fast obstacles) trajectory seen here



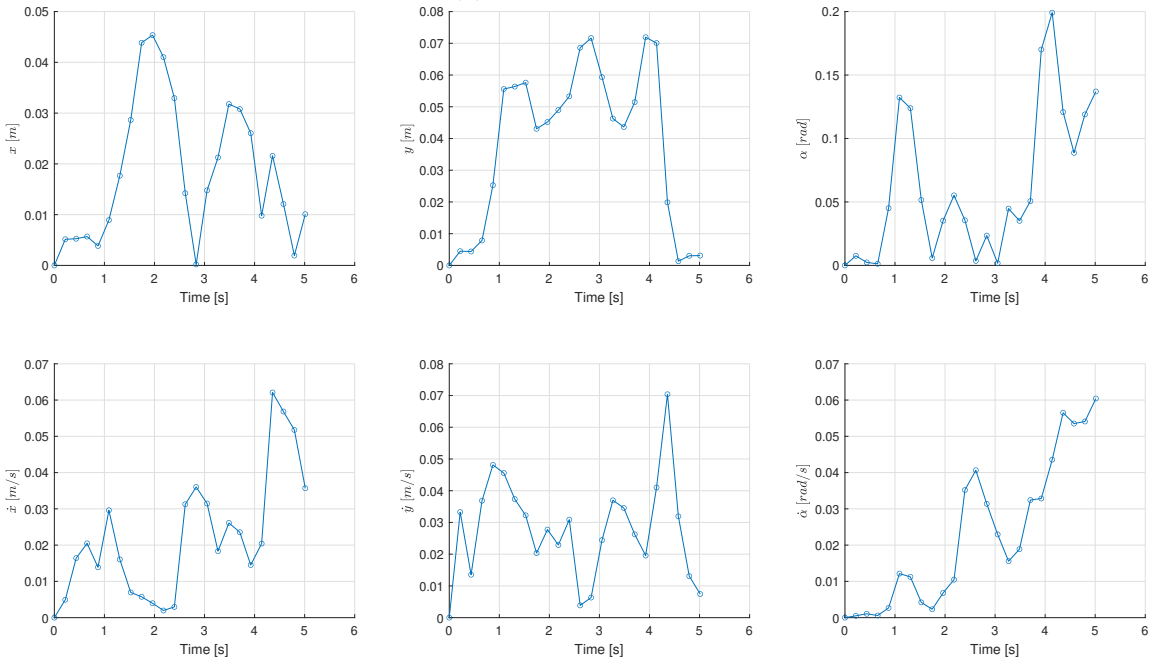
(a) $\boldsymbol{p}$-state variables



(b) Torques profile

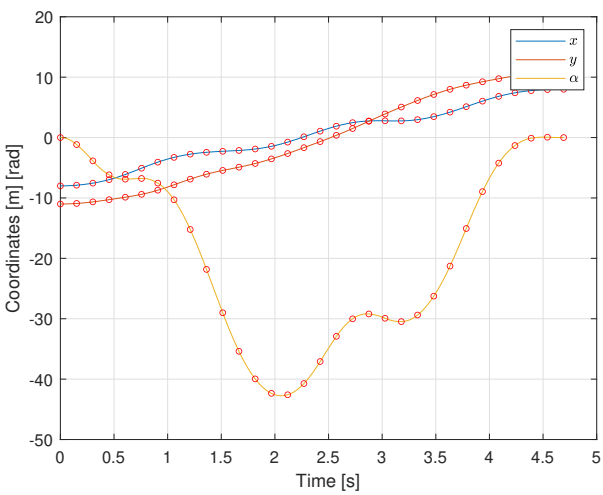


(c) Dynamical error

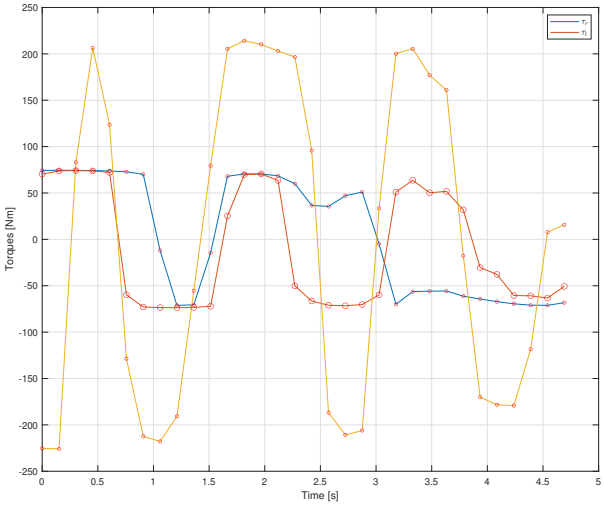


(d) Trajectory error

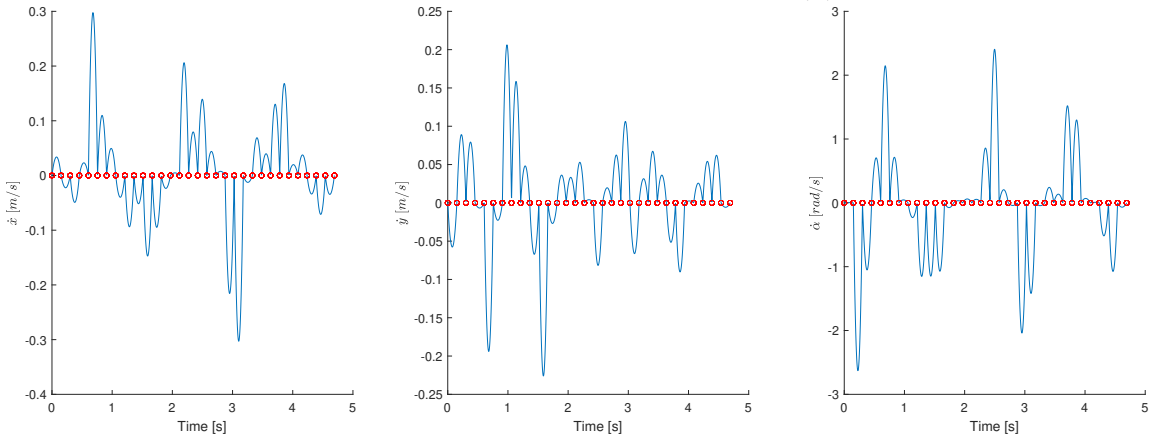
Figure B.7 – Plots corresponding to the dynamic crowded corridor (7 obstacles) trajectory seen here



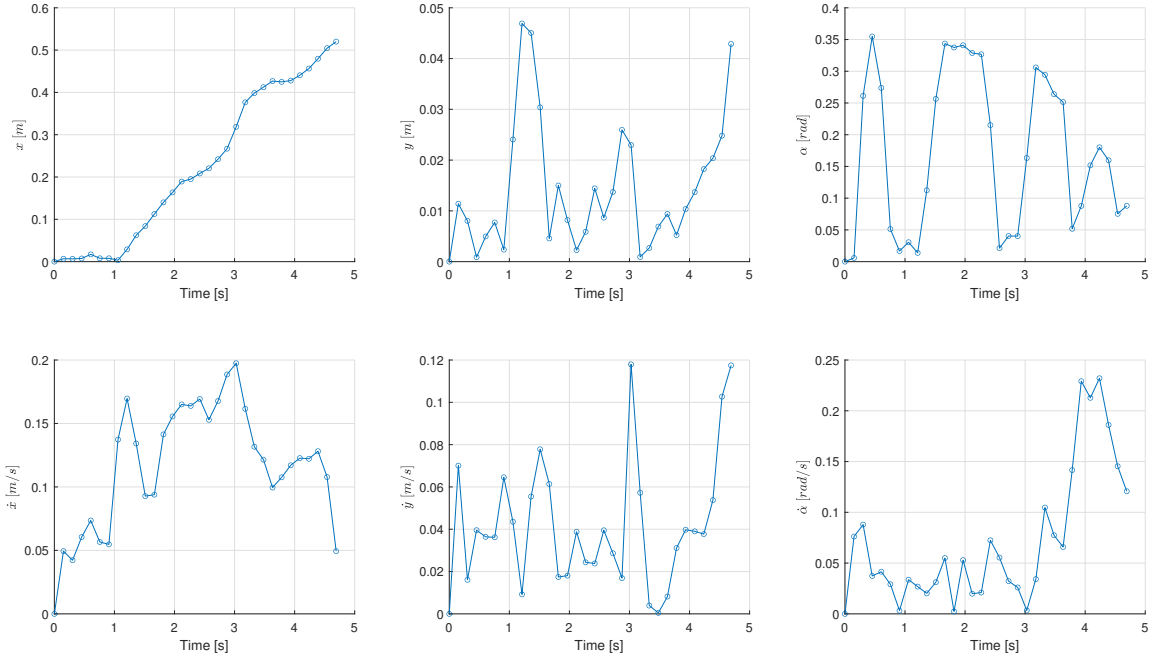
(a) $\boldsymbol{p}$-state variables



(b) Torque profile

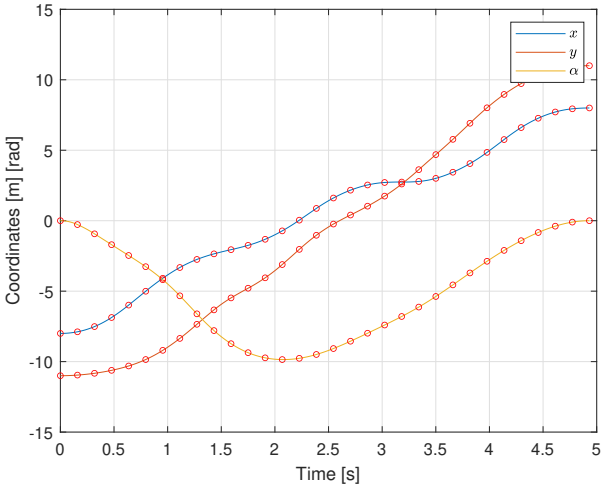


(c) Dynamical error

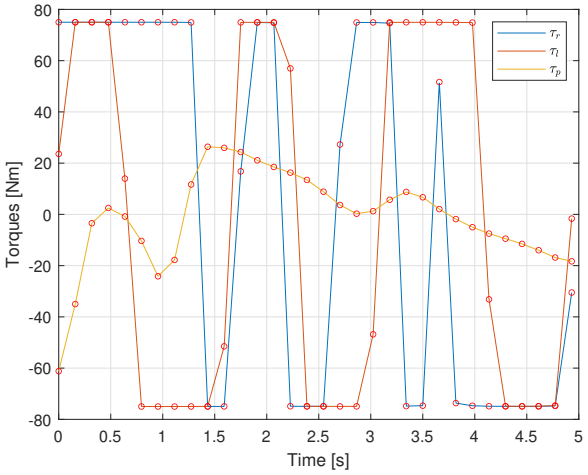


(d) Trajectory error

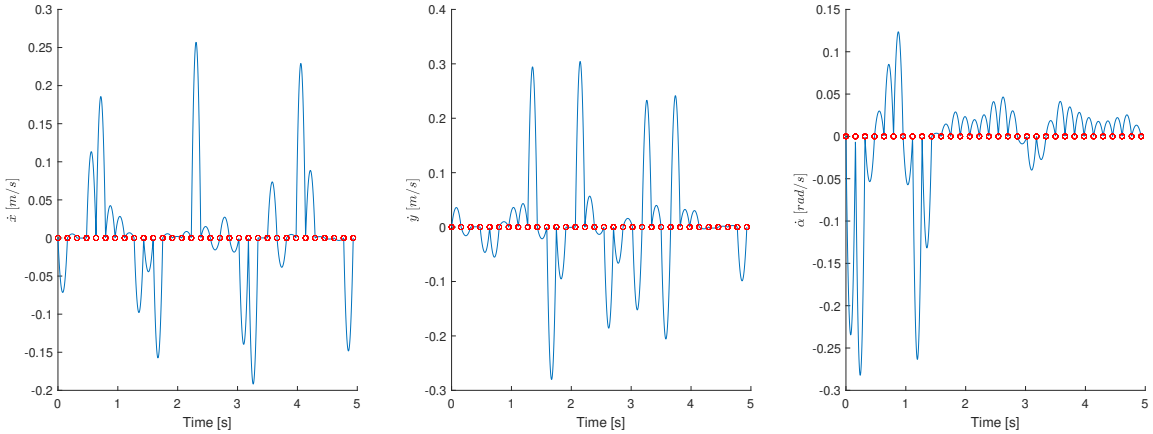
Figure B.8 – Plots corresponding to the insane box trajectory (time objective function) seen here



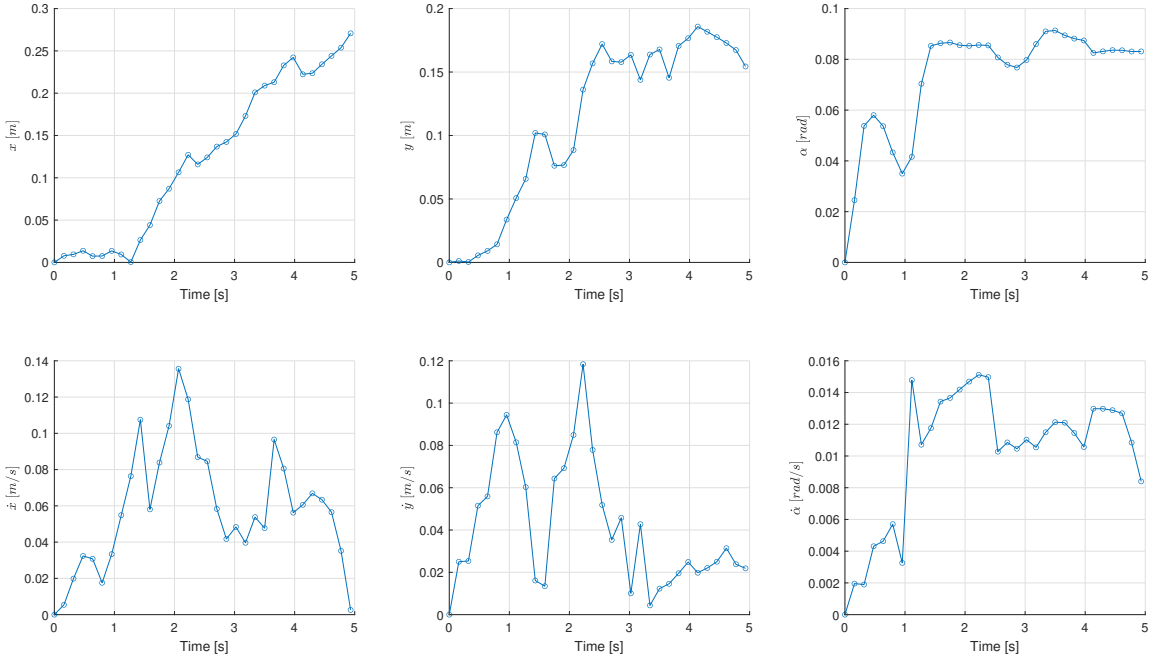
(a) $\boldsymbol{p}$-state variables



(b) Torque profile



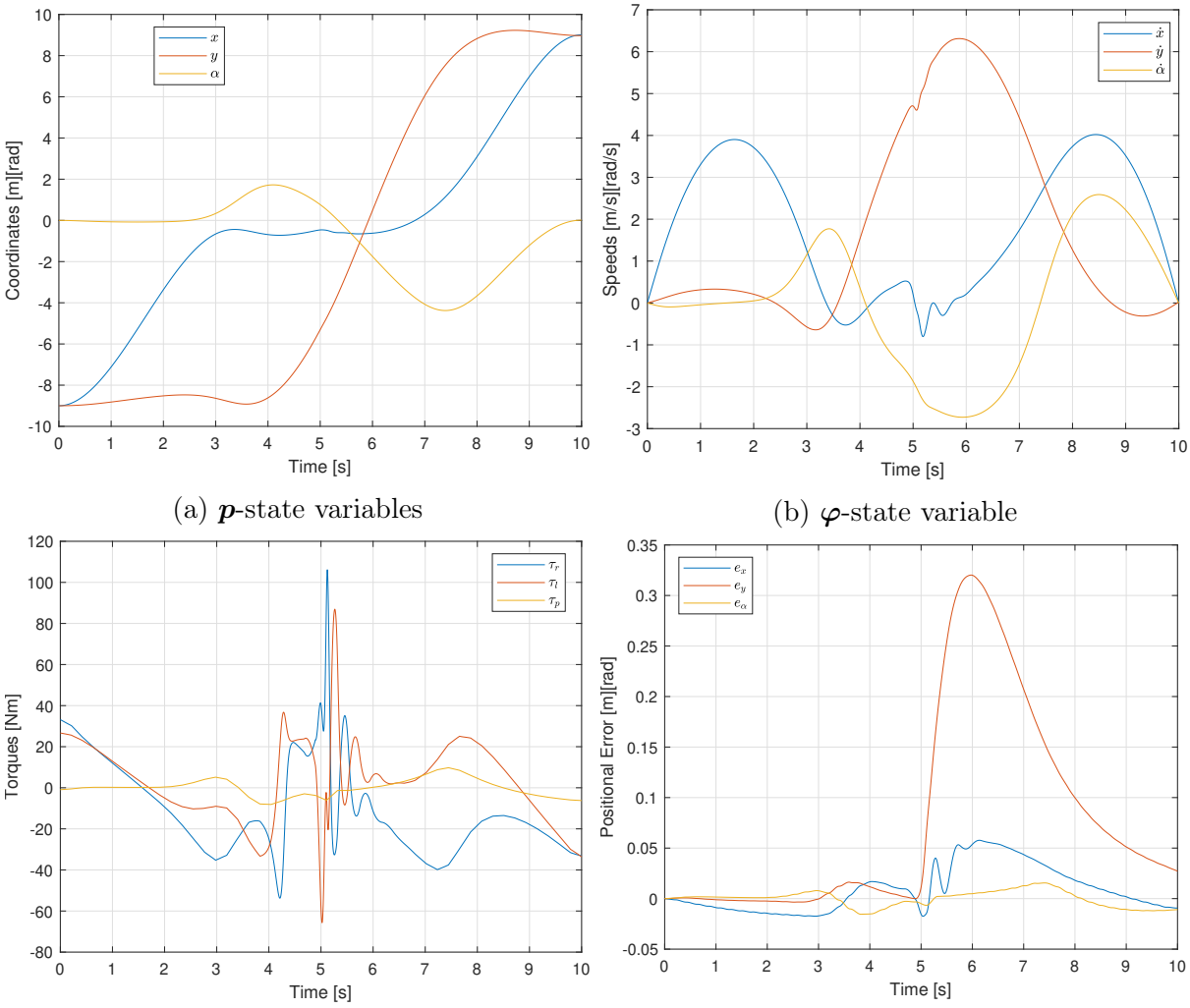
(c) Dynamical error



(d) Trajectory error

Figure B.9 – Plots corresponding to the insane box trajectory (time+forceterm objective function) seen here

B.2 Trajectory following using the controller



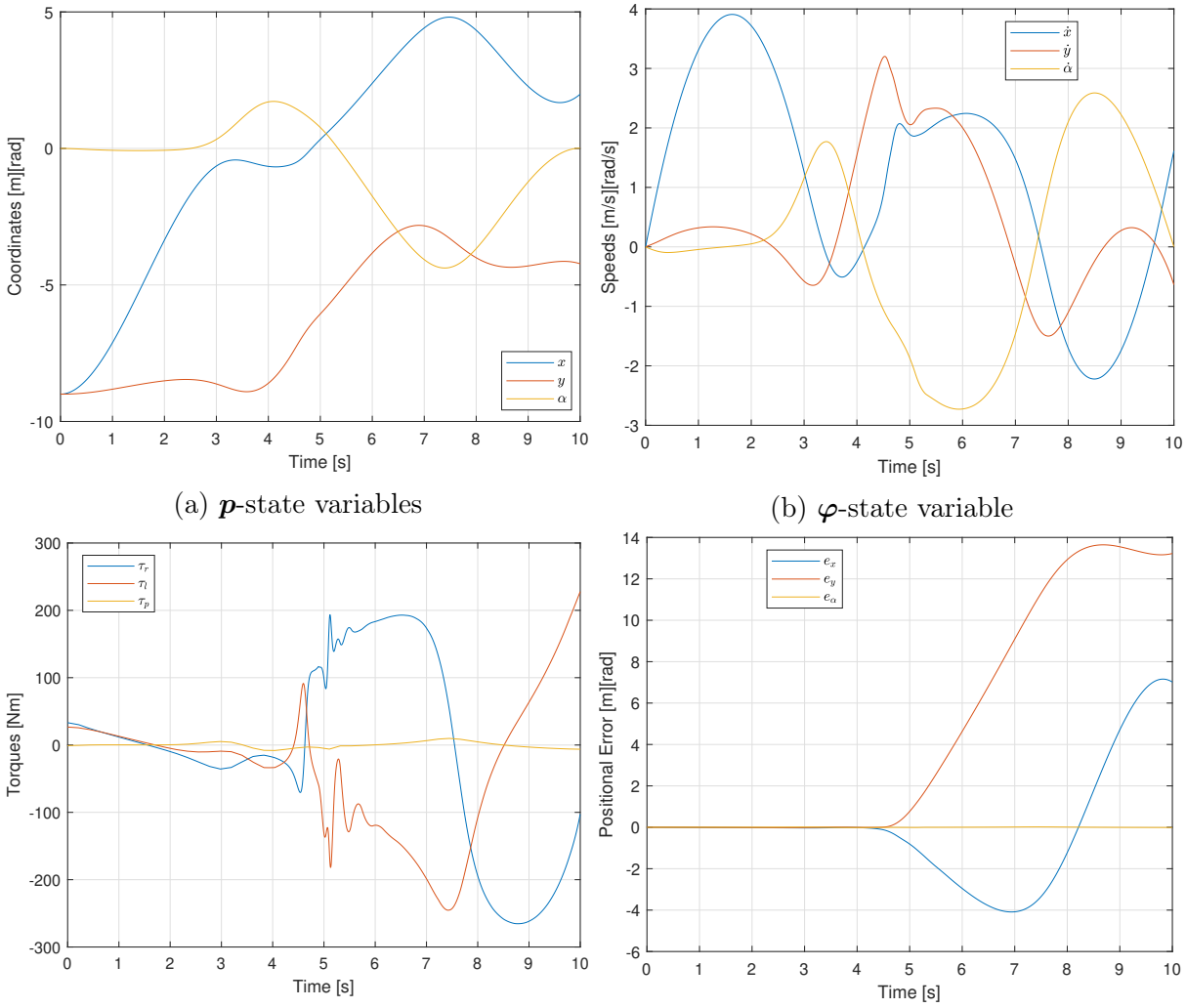
(a) $\boldsymbol{p}$-state variables

(b) $\boldsymbol{\varphi}$-state variable

(c) Torques profile

(d) Tracking error

Figure B.10 – Plots corresponding to the forced deviation trajectory following with a computed-torque controller seen here



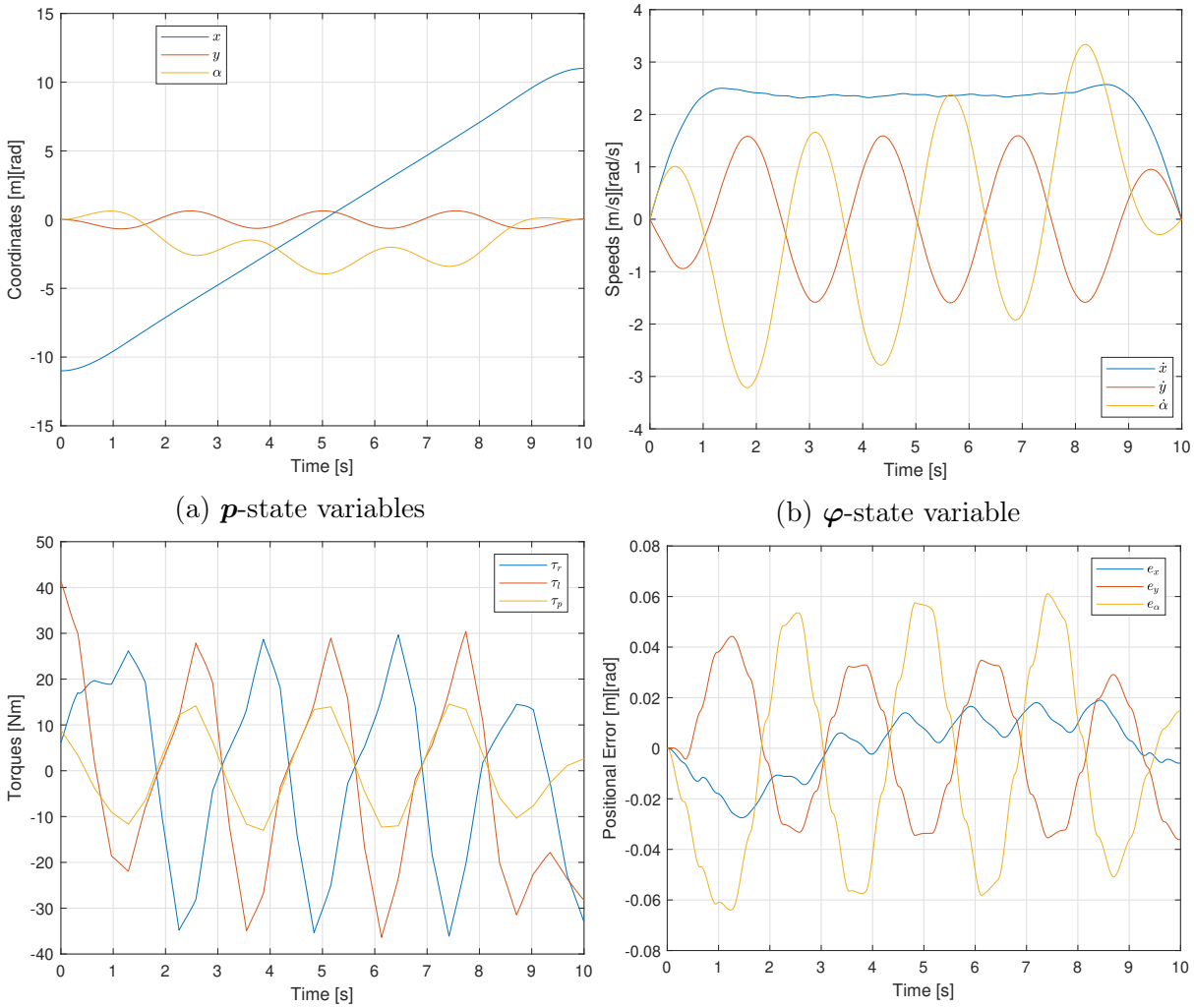
(a) $\boldsymbol{p}$-state variables

(b) $\boldsymbol{\varphi}$-state variable

(c) Torques profile

(d) Tracking error

Figure B.11 – Plots corresponding to the forced deviation trajectory following with interpolated control from optimizer seen here



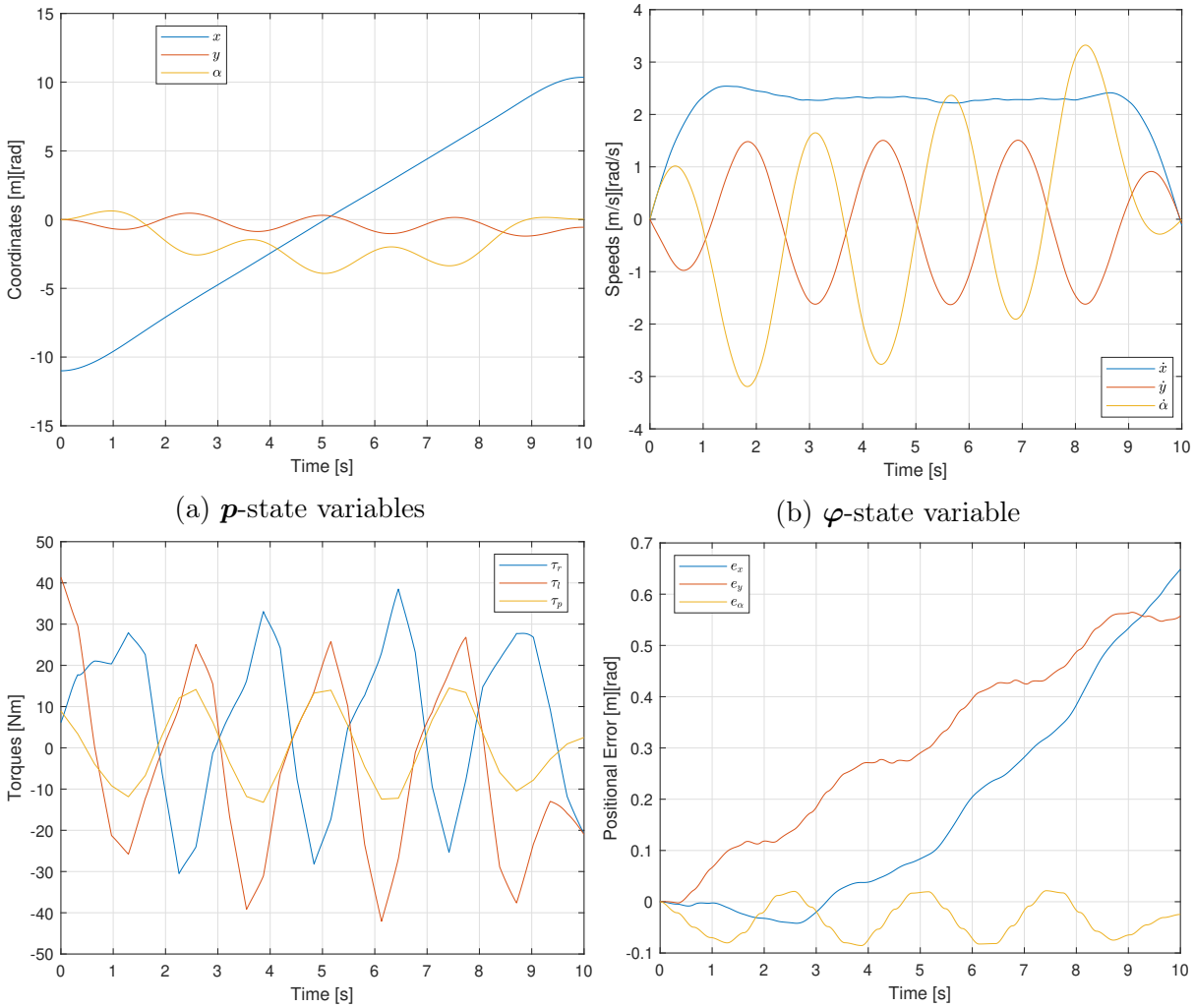
(a) $\boldsymbol{p}$-state variables

(b) $\boldsymbol{\varphi}$-state variable

(c) Torques profile

(d) Tracking error

Figure B.12 – Plots corresponding to the crowded corridor trajectory following with a computed-torque controller seen here



(a) $\boldsymbol{p}$-state variables

(b) $\boldsymbol{\varphi}$-state variable

(c) Torques profile

(d) Tracking error

Figure B.13 – Plots corresponding to the crowded corridor trajectory following with interpolated control from optimizer seen here