



DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Cooperative Manipulation without force/torque feedback: Control Design and Experiments

MATTEO MASTELLARO

Abstract

This thesis addresses the problem of the cooperative manipulation of a single object by N robotic agents. In particular, we propose two different task-space decentralized control protocols, in the sense that no online communication takes place between the agents. Moreover, no feedback on the contact forces/torques is required, therefore the use of corresponding sensors is avoided. Load sharing coefficient between the agents are employed to represent potential differences in power capabilities among the agents. In the former approach, each agent utilizes information associated with its own and the object's dynamics and quaternions are employed for the object's orientation, avoiding thus potential representation singularities. The latter methodology utilizes a model-free control protocol with prescribed transient and steady-state performance as well boundedness on the input magnitude and rate. The results are verified through realistic V-REP simulations and experimental studies with two WidowX robotic arms in both methodologies.

Contents

1	Introduction	1
2	Mathematical Model	3
2.1	Preliminaries	3
2.2	Kinematics	4
2.3	Dynamics	5
3	Quaternion-based Cooperative Manipulation without Force/Torque Information	9
3.1	Preliminaries	9
3.2	Non Adaptive Control Design	10
3.3	Adaptive Control Design	11
4	Prescribed Performance Control	13
4.1	Introduction to Prescribed Performance Control	13
4.2	Control design	13
4.3	Control parameters design	15
5	Hardware	17
5.1	WidowX mark II	17
5.2	Servos Characteristics	19
6	Simulation Environment	21
7	Code	23
7.1	Dynamic Matrices	23
7.2	Driver	26
7.3	Controllers	29
8	Simulation Results	31
8.1	Quaternion based approach	31
8.2	Prescribed Performance Control	37
9	Experimental Results	41
9.1	Quaternion based approach	42
9.2	Prescribed Performance Control	46
10	Conclusions	49
	Bibliography	51

Acknowledgments

All the work exposed in this thesis has been carried out during a semester abroad at KTH Royal Institute of Technology in Stockholm.

When it comes to thanking the people that supported me during this period and made this work possible I must cite my parents, for all the support they gave to me during all my university career.

Another special thanks goes to Dimos Dimarogonas, my examiner, for welcoming me in KTH and giving me this beautiful opportunity, and to Christos Verginis, my supervisor, for encouraging and advising me during this entire experience.

Finally I want to express all my gratitude to Antonio, Massimiliano, Dario, Pedro, Umarand Paul, for the friendship, and the good time spent together in Stockholm.

Chapter 1

Introduction

The problem of cooperative manipulation has occupied the research community for more than two decades and there exist a large variety of related works. Early solutions develop both centralized control architectures, as well as decentralized setups, where each agent compute its own control signals, either by communicating with each other or with no communication at all [1]- [7]. Impedance and force/motion control are the most used methodology in the related literature [1], [8]- [15]. Most of the aforementioned works employ force/torque sensors in order to acquire knowledge of the manipulator-object contact force/torques, which however, may result to performance decline due to sensor noise or mounting difficulties. Recently manipulator grippers that allow to rigidly grasp certain object has made appearance in the research scene [16]. By employing grasping rigidity we will see that such torque/force sensors are no more necessary.

In this thesis we are going to present three decentralized cooperative manipulation controllers that employs grasp rigidity in order to avoid using dedicated sensors to retrieve forces and/or torques. An important characteristic in robot manipulation is the representation of the agents and object orientation. Most common tools to represent orientations are rotation matrices, Euler angles and the angle/axis convention. Due to difficulty in retrieving an orientation error rotation matrices are rarely used in robotic problems. Moreover, as we will see, the mapping from Euler angles and angles/axis to angular velocities can exhibit singularity at certain points. For this reasons in the first controller implementation we will use quaternions to represent orientations, without complicating the control design. In addition most of the works in the relate literature consider known dynamic parameters regarding the object and the robotic agents. However, the accurate knowledge of such parameter (e.g. masses or moments of inertia) can be a challenging issue in [3] and [17] adaptive control schemes through gain tuning and robust pose regulation is applied to overcome the problem, while in [19] a model free approach is developed. In light of that we are going to extend our quaternion based approach to the adaptive solution and adapt, in another controller, the formulation in [19] to the cooperative manipulation case. Moreover all proposed controllers use load sharing coefficients in order to distribute the object payload according to the potentially different capabilities of the agents.

Finally, simulation and experimental results using two WidowX Mark II robotic arms validate the proposed methodologies.

Regarding the quaternion based controller and its extension to the adaptive case, the main novelty of our approach is the combination of *i*) control design in task-space variables which avoids explicit computation of inverse kinematics algorithms, *ii*) coupled object-agent dynamic formulation which does not require contact force/torque measurements, *iii*) the ex-

tension to an adaptive version, where the dynamic parameters of object and robots are considered unknown and *iv)* the employment of unit quaternions for the object orientation thus avoiding potential representation singularities.

Concerning the prescribe performance controller adaptation instead, to the best of the authors' knowledge, this is the first approach that integrates model-free cooperative manipulation under bounded inputs with prescribed transient and steady state performance in an experimental framework.

Chapter 2

Mathematical Model

2.1 Preliminaries

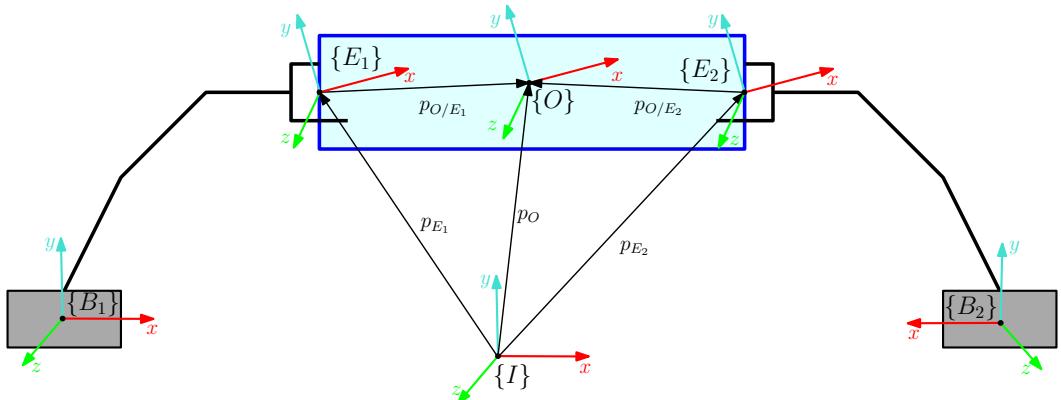


Figure 2.1: Two robotic arms rigidly grasping an object.

Consider N agents rigidly grasping an object as in figure 2.1. The following notations will be used:

- We will denote with $q_i \in \mathbb{R}^{n_i}$, $i \in [1\dots N] := \mathcal{N}$ the generalized joint-space variables of each agent.
- With E_i and B_i we will denote the arm's end effector and base link frames. With O the object's center of mass frame, and with I the inertial reference frame.
- The arms' reference frames for links and joints are computed according to the Denavit-Hartenberg convention [18].
- The vector connecting the origins of coordinate frames A and B expressed in frame C coordinates in 3D space is denoted as $p_{B/A}^C \in \mathbb{R}^3$.
- We denote as $\phi_{A/B} \in \mathbb{T}^3$ the Euler angles representing the orientation of B with respect to A , where \mathbb{T}^3 is the 3D torus. We also define the set $\mathbb{M}^n = \mathbb{R}^n \times \mathbb{T}^n$.
- The rotation matrix from A to B is denoted as $R_{B/A} \in SO(3)$, where $SO(3)$ is the 3D rotation group.

- The angular velocity of frame B with respect to A , expressed in C , is denoted as $\omega_{B/A}^C \in \mathbb{R}^3$ and it holds that $\dot{R}_{B/A} = S(\omega_{B/A}^A)R_{B/A}$ [18]. Where $S(a)$ is the skew-symmetric matrix defined according to $S(a)b = a \times b$.
- Finally for notational brevity, when a coordinate frame corresponds to the inertial reference frame I , we will omit its explicit notation. And all vector and matrix differentiations will be with respect to the inertial frame I , unless otherwise stated.

2.2 Kinematics

The end effectors positions $p_{E_i}^{B_i}(t)$ and orientations ϕ_{E_i/B_i} are easily computed from the agent's joint-space variables $q_i(t)$. From the arm's forward and differential kinematics, if we denote with $J_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{6 \times 6}$ the geometric Jacobian of agent i ; with $k_{p_i} : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^3$ and $k_{\phi_i} : \mathbb{R}^{n_i} \rightarrow \mathbb{T}^3$ its forward kinematics mapping, we will have that [18]:

$$p_{E_i}^{B_i}(t) = k_{p_i}(q_i(t)) \quad (2.1a)$$

$$\phi_{E_i/B_i}(t) = k_{\phi_i}(q_i(t)) \quad (2.1b)$$

$$v_{E_i}^{B_i}(t) = [\dot{p}_{E_i}^{B_i}(t)^T \quad \omega_{E_i}^{B_i}(t)^T]^T = J_i(q_i(t))\dot{q}_i \quad (2.1c)$$

So:

$$p_{E_i}(t) = p_{B_i} + R_{B_i}(\phi_{B_i})p_{E_i}^{B_i}(t) \quad (2.2a)$$

$$\phi_{E_i}(t) = \phi_{E_i/B_i}(t) + \phi_{B_i} \quad (2.2b)$$

$$v_{E_i}(t) = R_{B_i}(\phi_{B_i})v_{E_i}^{B_i}(t) \quad (2.2c)$$

Given the rigid grasping the vectors $p_{E_i/O}^{E_i}$ and $\phi_{E_i/O}$ are constant, thus they can be computed offline and passed to each agent. From this value each arm can compute the object's pose and orientation with just the information in it's joint-space variables:

$$p_O(t) = p_{E_i}(t) - R_{E_i}(q_i(t))p_{E_i/O}^{E_i} \quad (2.3a)$$

$$\phi_O(t) = \phi_{E_i}(t) - \phi_{E_i/O} \quad (2.3b)$$

Differentiation of (2.3a) along with the fact that, due to the grasping rigidity, it holds that $\omega_O = \omega_{E_i}$, allow us to compute the object velocity:

$$\begin{aligned} \dot{p}_O(t) &= \dot{p}_{E_i}(t) - \dot{R}_{E_i}(q_i(t))p_{E_i/O}^{E_i} \\ &= \dot{p}_{E_i}(t) - S(\omega_{E_i}(t))R_{E_i}(q_i(t))p_{E_i/O}^{E_i} \\ &= \dot{p}_{E_i}(t) - S(\omega_{E_i}(t))p_{O/E_i}(q_i(t)) \\ &= \dot{p}_{E_i}(t) + S(p_{E_i/O}(q_i(t)))\omega_{E_i}(t) \end{aligned}$$

Thus:

$$v_o(t) = \begin{bmatrix} \dot{p}_O(t) \\ \omega_O(t) \end{bmatrix} = \begin{bmatrix} I_3 & S(p_{E_i/O}(q_i(t))) \\ 0_{3 \times 3} & I_3 \end{bmatrix} \begin{bmatrix} \dot{p}_{E_i}(t) \\ \omega_{E_i}(t) \end{bmatrix} = J_{i_O}(q_i(t))v_{E_i}(t) \quad (2.4)$$

Where $v_{E_i} = [\dot{p}_{E_i}^T, \omega_{E_i}^T]^T$ is the end effector's linear and angular velocities and $J_{i_O} : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{6 \times 6}$ is a smooth mapping representing the agent-to-object Jacobian matrix.

Moreover if we define the vector $x_o(t) = [p_O^T(t), \phi_O^T(T)]^T$ with $\phi_o(t) = [\gamma_o(t), \theta_o(t), \psi_o(t)]^T$ being the Euler angle representation of the object orientation, we will have:

$$\dot{x}_o(t) = J_{o_r}^{-1}(x_o(t))v_o(t), \quad (2.5)$$

where $J_{o_r} : \mathbb{M} \rightarrow \mathbb{R}^{6 \times 6}$ is the object representation jacobian $J_{o_r}(x_o) = \text{diag}\{I_3, J_{o_r,\theta}(x_o)\}$ with:

$$J_{o_r,\theta}(x_o) = \begin{bmatrix} 1 & 0 & \sin(\theta_o) \\ 0 & \cos(\gamma_o) & -\cos(\theta_o)\sin(\gamma_o) \\ 0 & \sin(\gamma_o) & -\cos(\theta_o)\cos(\gamma_o) \end{bmatrix} \quad (2.6)$$

Note that $J_{o_r}(x_o)$ is singular in $\theta_o = \pm \frac{1}{2}$, i.e. all the positions where the object is oriented vertically w.r.t. the inertial reference frame, we will then avoid such configurations.

Remark 2.2.1. Notice that J_{i_O} is always full rank due to grasp rigidity, thus the object-to-agent Jacobian matrix can always be directly derived from J_{i_O} :

$$J_{O_i}(q_i) = J_{i_O}^{-1}(q_i) = \begin{bmatrix} I_3 & -S(p_{E_i/O}(q_i)) \\ 0_{3 \times 3} & I_3 \end{bmatrix} = \begin{bmatrix} I_3 & S(p_{O/E_i}(q_i)) \\ 0_{3 \times 3} & I_3 \end{bmatrix}. \quad (2.7)$$

2.3 Dynamics

Agents' Dynamics

The joint-space dynamics of agent i can be computed using the Lagrangian formulation [18]:

$$B_i(q_i)\ddot{q}_i + N_i(q_i, \dot{q}_i)\dot{q}_i + p_i(q_i) + f_{q_i}(q_i, \dot{q}_i) + w_{q_i}(t) = \tau_i - J_i^T \lambda_i \quad (2.8)$$

Where $B_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_i \times n_i}$ is the joint-space positive definite inertia matrix, $N_i : \mathbb{R}^{n_i} \times \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_i \times n_i}$ is the joint-space Coriolis matrix, $p_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_i}$ is the joint-space gravity vector, $f_{q_i} : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_i}$ is a model field representing model uncertainties and $w_{q_i} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{n_i}$ is a bounded vector representing external disturbances. In the right side instead, $\lambda_i \in \mathbb{R}^6$ is the generalized force vector that agent i exerts on the object and $\tau_i \in \mathbb{R}^{n_i}$ is the vector of generalized joint-space inputs, with $\tau_i = [\lambda_{B_i}^T, \tau_{\alpha_i}^T]^T$, where $\lambda_{B_i} = [f_{B_i}^T, \mu_{B_i}^T]^T \in \mathbb{R}^6$ is the generalized force vector on the center of mass of the agent's base and $\tau_{\alpha_i} \in \mathbb{R}^6$ is the torque inputs of the robotic arms' joints.

Now, by inverting (2.8) and using (2.1c) and its derivative, we can obtain the task-space agent dynamics [18]:

$$M_i(q_i)\ddot{v}_{E_i} + C_i(q_i, \dot{q}_i)v_{E_i} + g_i(q_i) + f_{q_i}(q_i, \dot{q}_i) + w_{q_i}(t) = u_i - \lambda_i \quad (2.9)$$

Where $M_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{6 \times 6}$ is the positive definite inertia matrix, $C_i : \mathbb{R}^{n_i} \times \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{6 \times 6}$ is the Coriolis matrix, $g_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^6$ is the task-space gravity term, $f_i \in \mathbb{R}^6$ is the vector of generalized forces that agent i exerts on the grasping point with the object and finally u_i is the task-space wrench acting as control input.

From the kineto-static duality [18] it is possible to derive the input joint torques from u_i as:

$$\tau_i = J_i^T u_i + (I_{n_i} - J_i^T(q_i)J_i^{T\dagger}(q_i))\tau_{i_0} \quad (2.10)$$

where the term τ_{i_0} has been introduced to take account of over actuated agents, and does not contribute to the end-effector forces.

Finally, given that q_i is not a position of singularity for $J_i(q_i)$, the task-space matrices can be directly derived from the joint-space ones:

$$M_i = (J_i B_i^{-1} J_i^T)^{-1} \quad (2.11a)$$

$$C_i = M_i (J_i B_i^{-1} N_i - \dot{J}_i) J_i^{-1} \quad (2.11b)$$

$$g_i = M_i J_i B_i^{-1} p_i \quad (2.11c)$$

Object's Dynamics

The following second order dynamics for the object can be derived based on the Newton-Euler formulation:

$$M_o(x_o)\ddot{v}_o + C_o(x_o, v_o)v_o + g_o(x_o) + w_o(t) = \lambda_o \quad (2.12)$$

where $x_o = [p_O^T, \phi_O^T]^T : \mathbb{R}_{\geq 0} \rightarrow \mathbb{M}$, $M_o : \mathbb{M} \rightarrow \mathbb{R}^{6 \times 6}$ is the positive definite inertia matrix, $C_o : \mathbb{M} \times \mathbb{R}^6 \rightarrow \mathbb{R}^{6 \times 6}$ is the Coriolis matrix, $g_o : \mathbb{M} \rightarrow \mathbb{R}^6$ is the gravity vector, $w_o : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$ is a bounded vector representing external disturbances and finally $\lambda_o \in \mathbb{R}^6$ is the vector of generalized forces acting on the object's center of mass.

If $\lambda_i, i \in \mathcal{N}$ is the force that each agent exert on the object, exploiting as before the kineto-static duality and considering the rigid grasping we have:

$$\lambda_o = G^T(q)\lambda \quad (2.13)$$

where $\lambda = [\lambda_i^T]_{i \in \mathcal{N}}^T \in \mathbb{R}^{6N}$, $G(q) = [J_{o_1}^T(q_1), \dots, J_{o_N}^T(q_N)]^T$, $G : \mathbb{R}^n \rightarrow \mathbb{R}^{6N \times 6}$ is the grasping matrix, and $q = [q_i^T]_{i \in \mathcal{N}}^T \in \mathbb{R}^n$, $n = \sum_{i \in \mathcal{N}} n_i$ is the vector containing all agents' joint variables.

Remark 2.3.1. There might be internal forces $\lambda_I \in \mathbb{R}^{6N}$ generated by the agents that doesn't contribute in the object movement, hence $G^T(q)\lambda_I = 0$. If we denote with $\lambda_M \in \mathbb{R}^{6N}$ the forces that actually produces a movement for the object we will have [20]:

$$\begin{aligned} \lambda &= \lambda_I + \lambda_M \\ \lambda_o &= G^T(q)\lambda = G^T(q)(\lambda_I + \lambda_M) \\ &= G^T(q)\lambda_M \end{aligned}$$

so:

$$\lambda_M = \frac{1}{N} G^*(q) G^T(q) \lambda \quad (2.14)$$

and finally:

$$\lambda_I = \lambda - \lambda_M = \lambda - \frac{1}{N} G^*(q) G^T(q) \lambda = \left(I_{6N} - \frac{1}{N} G^*(q) G^T(q) \right) \lambda. \quad (2.15)$$

Where $\frac{1}{N} G^*(q)$ is a generalized inverse of $G^T(q)$ [20]:

$$G^*(q) = \begin{bmatrix} J_{o_1}^{-T}(q) \\ \vdots \\ J_{o_N}^{-T}(q) \end{bmatrix} \quad (2.16)$$

Coupled Dynamics

Our goal is now to compute a comprehensive dynamic equation for the entire system, i.e. object plus agents. The agents dynamics (2.9) can be written in vector form as:

$$M(q)\dot{v} + C(q, \dot{q})v + g(q) + f(q) + w(t) = u - \lambda \quad (2.17)$$

where, for simplicity, we omitted the disturbance and model uncertainties term, q and λ are defined as in the previous section, $v = \left[[v_i^T]_{i \in \mathcal{N}} \right]^T \in \mathbb{R}^{6N}$, $M = \text{diag} \{ [M_i]_{i \in \mathcal{N}} \} \in \mathbb{R}^{6N \times 6N}$, $C = \text{diag} \{ [C_i]_{i \in \mathcal{N}} \} \in \mathbb{R}^{6N \times 6N}$, $u = \left[[u_i^T]_{i \in \mathcal{N}} \right]^T \in \mathbb{R}^{6N}$, $g = \left[[g_i^T]_{i \in \mathcal{N}} \right]^T \in \mathbb{R}^{6N}$, $f = \left[[f_i^T]_{i \in \mathcal{N}} \right]^T \in \mathbb{R}^{6N}$ and $w = \left[[w_i^T]_{i \in \mathcal{N}} \right]^T \in \mathbb{R}^{6N}$.

We can now compute the overall dynamics. By substituting (2.17) in (2.13) we obtain:

$$\lambda_o = G^T(q) (u - M(q)\dot{v} - C(q, \dot{q})v - g(q)). \quad (2.18)$$

By differentiation of (2.4) we get:

$$\dot{v}_i = \dot{J}_{o_i}(q_i, \dot{q}_i)v_o(t) + J_{o_i}(q_i)\dot{v}_o. \quad (2.19)$$

Finally, by substituting (2.4), (2.19) and (2.12) in (2.18) we obtain the overall coupled dynamics:

$$\bar{M}(q, x_o)\dot{v}_o + \bar{C}(q, \dot{q}, x_o, v_o)v_o + \bar{g}(q, x_o) + \bar{w}(q, t) = G^T(q)u, \quad (2.20)$$

where:

$$\bar{M} = M_o + G^T M G \quad (2.21)$$

$$\bar{C} = C_o + G^T C G + G^T M \dot{G} \quad (2.22)$$

$$\bar{g} = g_o + G^T g + G^T f \quad (2.23)$$

$$\bar{w} = w_o + G^T w \quad (2.24)$$

Chapter 3

Quaternion-based Cooperative Manipulation without Force/Torque Information

3.1 Preliminaries

In order to avoid representation singularities in (2.4) we are going to use quaternions to express frames orientations. Given two frames $\{A\}$ and $\{B\}$ we define a unit quaternion $\xi_{B/A} = [\eta_{B/A}, \varepsilon_{B/A}^T]^T \in \mathcal{S}^3$, where with \mathcal{S}^n we denote the n dimension sphere, $\eta_{B/A} \in \mathbb{R}$ and $\varepsilon_{B/A} \in \mathcal{S}^2$. The quaternion subjects to the constraint $\eta_{B/A}^2 + \varepsilon_{B/A}^T \varepsilon_{B/A} = 1$. Given a rotation between two frames of θ degrees about the vector γ , $\|\gamma\|_2 = 1$, and the corresponding rotation matrix $R_{B/A} = [r_{ij}]$, $i, j \in \{1, 2, 3\}$, we can compute the quaternion directly from θ and γ or from the rotation matrix as follow [18]:

$$\eta_{B/A} = \cos\left(\frac{\theta}{2}\right) = \frac{1}{2}\sqrt{r_{11} + r_{22} + r_{33} + 1} \quad (3.1)$$

$$\varepsilon_{B/A} = \sin\left(\frac{\theta}{2}\right)\gamma = \frac{1}{2}\begin{bmatrix} \text{sign}(r_{32} - r_{23})\sqrt{r_{11} - r_{22} - r_{33} + 1} \\ \text{sign}(r_{32} - r_{23})\sqrt{r_{11} - r_{22} - r_{33} + 1} \\ \text{sign}(r_{21} - r_{12})\sqrt{r_{33} - r_{22} - r_{11} + 1} \end{bmatrix}. \quad (3.2)$$

For a given quaternion $\xi_{B/A} = [\eta_{B/A}, \varepsilon_{B/A}^T]^T \in \mathcal{S}^3$ it's conjugate, that corresponds to the orientation of frame $\{A\}$ with respect to $\{B\}$, is $\xi_{B/A}^* = \xi_{A/B} = [\eta_{B/A}, -\varepsilon_{B/A}^T]^T \in \mathcal{S}^3$. Moreover given two quaternions $\xi_i = [\eta_i, \varepsilon_i^T]^T$, $i \in \{1, 2\}$, the quaternion product is defined as:

$$\xi_1 \otimes \xi_2 = \begin{bmatrix} \eta_1\eta_2 - \varepsilon_1^T \varepsilon_2 \\ \eta_1\varepsilon_2 + \eta_2\varepsilon_1 + S(\varepsilon_1)\varepsilon_2 \end{bmatrix}, \quad (3.3)$$

it's time derivative as:

$$\dot{\xi}_{B/A} = \frac{1}{2}E(\xi_{B/A})\omega_{B/A}^A, \quad (3.4)$$

where:

$$E(\xi) = \begin{bmatrix} -\varepsilon^T \\ \eta I_3 - S(\varepsilon) \end{bmatrix}, \quad (3.5)$$

and finally it can be shown that $E^T(\xi)E(\xi) = I_3$ and hence [18]:

$$\omega_{B/A}^A = 2E^T(\xi_{B/A})\dot{\xi}_{B/A}. \quad (3.6)$$

3.2 Non Adaptive Control Design

Given a bounded target position and orientation for the object, specified by $p_{O,d}(t) \in \mathbb{R}^3$ and $\xi_{O,d}(t) \in \mathcal{S}^3$ with bounded first and second derivative, the goal is to find u in (2.20) such that:

$$\lim_{t \rightarrow \infty} \begin{bmatrix} p_O(t) \\ \xi_O(t) \end{bmatrix} = \begin{bmatrix} p_{O,d}(t) \\ \xi_{O,d}(t) \end{bmatrix}. \quad (3.7)$$

First we will need to define the errors associated with the object pose and desired pose trajectory.

Lets define as position error the vector $e_p : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^3$:

$$e_p(t) = p_O(t) - p_{O,d}(t), \quad (3.8)$$

since quaternion do not form a vector space we need to use its property to extract the orientation error. If $e_\xi = [e_\eta, e_\varepsilon^T]^T : \mathbb{R}_{\geq 0} \rightarrow \mathcal{S}^3$ is the quaternion describing the orientation error, we will have that [18]:

$$\begin{aligned} e_\xi(t) &= \begin{bmatrix} e_\eta(t) \\ e_\varepsilon(t) \end{bmatrix} = \xi_{O,d} \otimes \xi_O^*(t) = \begin{bmatrix} \eta_{o,d}(t) \\ \varepsilon_{o,d} \end{bmatrix} \otimes \begin{bmatrix} \eta_o(t) \\ -\varepsilon_o \end{bmatrix} \\ &= \begin{bmatrix} \eta_o(t)\eta_{o,d}(t) + \varepsilon_o^T \varepsilon_{o,d} \\ \eta_o(t)\varepsilon_{o,d}(t) - \eta_{o,d}(t)\varepsilon_o + S(\varepsilon_o)(t)\varepsilon_{o,d}(t) \end{bmatrix} \end{aligned} \quad (3.9)$$

By taking the time derivative of (3.8) and (3.9), employing (3.4) and (3.6) and certain properties of skew-simmetric matrices it can be shown that [18]:

$$\dot{e}_p(t) = \dot{p}_O(t) - \dot{p}_{O,d}(t) \quad (3.10a)$$

$$\dot{e}_\eta(t) = \frac{1}{2} e_\varepsilon^T(t) e_\omega(t) \quad (3.10b)$$

$$\dot{e}_\varepsilon(t) = -\frac{1}{2} (\eta_o(t)I_3 + S(e_\varepsilon(t))e_\omega(t) - S(e_\varepsilon(t))\omega_{O,d}) \quad (3.10c)$$

where $e_\omega : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^3$, $e_\omega(t) = \omega_O(t) - \omega_{O,d}(t)$ and $\omega_O(t) = 2E^T(\xi_O)\dot{\xi}_O(t)$.

Notice that, considering the properties of unit quaternions, when $\xi_O = \pm \xi_{O,d}$, $e_\xi(t) = [\pm 1 \ 0_{1 \times 3}]^T$, therefore our objective will be to have:

$$\lim_{t \rightarrow \infty} \begin{bmatrix} e_p(t) \\ |e_\eta(t)| \\ e_\varepsilon(t) \end{bmatrix} = \begin{bmatrix} 0_{3 \times 1} \\ 1 \\ 0_{3 \times 1} \end{bmatrix} \quad (3.11)$$

We can now define the reference velocity signal:

$$v_o^r(t) = \begin{bmatrix} \dot{p}_o^r(t) \\ \omega_o^r(t) \end{bmatrix} = \begin{bmatrix} \dot{p}_{o,d}(t) - k_p e_p(t) \\ \omega_{o,d}(t) + k_\varepsilon e_\varepsilon(t) \end{bmatrix} = v_{o,d}(t) - K e(t) \quad (3.12)$$

where $v_{o,d}(t) = [\dot{p}_{o,d}^T(t), \ \omega_{o,d}^T(t)]^T \in \mathbb{R}^6$, $K = \text{diag}\{kpI_3, \ k_\varepsilon I_3\} \in \mathbb{R}_{\geq 0}^{6 \times 6}$, $k_\varepsilon, k_p \in \mathbb{R}_{\geq 0}$ and $e(t) = [e_p^T(t), \ -e_\varepsilon^T(t)]^T \in \mathbb{R}^3 \times \mathcal{S}$.

We can now compute the velocity error $e_v : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$:

$$e_v(t) = v_o(t) - v_o^r(t), \quad (3.13)$$

and finally the decentralized control law for $u_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$, $i \in \mathcal{N}$ in (2.20):

$$u_i(t) = \mu_i(t) + f_{i,d}(t) \quad (3.14a)$$

$$\mu_i(t) = g_i + (C_i J_{o_i} + M_i \dot{J}_{o_i}) v_o^r(t) + M_i J_{o_i} \dot{v}_o^r(t) - J_{o_i}^{-T} (K_v e_v(t) + c_i e(t)) \quad (3.14b)$$

$$f_{i,d}(t) = c_i J_{o_i}^{-T} (M_o \dot{v}_o^r(t) + C_o v_o^r(t) + g_o) \quad (3.14c)$$

where $K_v = \text{diag}\{k_{v_i}\}_{i=1,2,\dots,6} \in \mathbb{R}_{\geq 0}^{6 \times 6}$ is a diagonal matrix gain.

As seen in section 2.3 in order to compute the M_i , C_i and g_i matrices, $i = 1, 2, O$, we will need all the dynamic parameters of robots and object (inertias moments, masses and geometric parameters). Those values are not trivial to compute, and often only an approximation can be retrieved. For this reason an adaptive version of the controller is presented in the next section.

3.3 Adaptive Control Design

Let's suppose to not know the dynamical parameters of robots and object. It can be shown [18] that the system dynamics can be written in the form:

$$M_i(q_i) \dot{v}_i + C_i(q_i, \dot{q}_i) v_i + g_i(q_i) = H_i(q_i, \dot{q}_i, v_i, \dot{v}_i) \theta_i \quad (3.15)$$

$$M_O(x_O) \dot{v}_O + C_O(x_O, \dot{x}_O) v_O + g_O(x_O) = Y_O(x_O, \dot{x}_O, v_O, \dot{v}_O) \theta_O \quad (3.16)$$

Where $\theta_O \in \mathbb{R}^{l_O}$ and $\theta_i \in \mathbb{R}^l$ are vectors of unknown but constant dynamic parameters of the object and agents respectively, and $H_i \in \mathbb{R}^{6 \times l}$, $Y_O \in \mathbb{R}^{6 \times l_O}$ are known regressor matrices not depending on the dynamic parameters of agents and object. Notice that l and l_O are not unique and depends on the specific factorization method used.

Since, as seen in section 2.2, J_{O_i} does not depends on θ_i and θ_O from (3.14b) and (3.15) we can write:

$$J_{O_i}^T M_i J_{O_i} \dot{v}_i + (J_{O_i}^T M_i \dot{J}_{O_i} + J_{O_i}^T C_i J_{O_i}) v_i + J_{O_i}^T g_i = Y_i(q_i, \dot{q}_i, v_i, \dot{v}_i) \theta_i \quad (3.17)$$

Where $Y_i \in \mathbb{R}^{6 \times l}$ is another regressor matrix independent of θ_i and θ_O . Hence from (3.14a), (3.14b), (3.14c) and (3.17) we can write the new decentralized control law as:

$$u_i(t) = J_{O_i}^{-T} \left(Y_i(q_i, \dot{q}_i, v_O^r, \dot{v}_O^r) \hat{\theta}_i(t) - c_i e(t) - k_{v_i} e_v(t) + c_i Y_O(x_O, \dot{x}_O, v_O^r, \dot{v}_O^r) \hat{\theta}_O^i(t) \right) \quad (3.18)$$

where $\hat{\theta}_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^l$ and $\hat{\theta}_O^i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{l_O}$ are the estimates of θ_i and θ_O respectively. The estimation is initialized with l_O and l values computed for example assuming the center of mass of each link to be in it's middle, and updated using the following adaptation laws:

$$\dot{\hat{\theta}}_O^i(t) = -\gamma_o Y_O^T(x_O, \dot{x}_O, v_O^r, \dot{v}_O^r) e_v(t) \quad (3.19a)$$

$$\dot{\hat{\theta}}_i(t) = -\gamma_i Y_i^T(q_i, \dot{q}_i, v_O^r, \dot{v}_O^r) e_v(t) \quad (3.19b)$$

where $\gamma_o \in \mathbb{R}_{\geq 0}$ and $\gamma_i \in \mathbb{R}_{\geq 0}$ are two constant parameters that needs to be experimentally calibrated and $e_v(t)$ is the signal defined in (3.13).

Chapter 4

Prescribed Performance Control

Even though the controller we talked about in the previous section works well in most scenarios, the computational level requested to implement it in complex scenes is significantly high. Moreover errors in the dynamical parameters of the model may affect the controller performance. A good alternative is that of prescribed performance control [19], which is adapted in this work in order to achieve predefined transient and steady state response bounds for the errors without relying on the specific state space model.

4.1 Introduction to Prescribed Performance Control

Prescribed performance control [19], describes the behavior where a tracking error $e(t) : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$ evolves strictly within a predefined region bounded by specific functions of time, achieving prescribed transient and steady state performance. We will have then:

$$-\rho_L(t) < e(t) < \rho_U(t), \quad \forall t \in \mathbb{R}_{\geq 0}, \quad (4.1)$$

where $\rho_L(t)$ and $\rho_U(t)$ are smooth bounded decay functions of time satisfying $\lim_{t \rightarrow \infty} \rho_L(t) > 0$ and $\lim_{t \rightarrow \infty} \rho_U(t) > 0$, called performance functions. If we take for example the exponential performance function we will have $\rho_i(t) = (\rho_i^0 - \rho_i^\infty)e^{-l_i t} + \rho_i^\infty$ with $\rho_i^0, \rho_i^\infty, l_i \in \mathbb{R}_{\geq 0}, i \in \{U, L\}$, appropriately chosen constants holding:

$$\begin{aligned} \rho_i(0) &= \rho_i^0 \\ \lim_{t \rightarrow \infty} \rho_i(t) &= \rho_i^\infty. \end{aligned}$$

The initial values ρ_L^0 and ρ_U^0 has to be chosen such that $-\rho_L^0 < e(0) < \rho_U^0$. ρ_L^∞ and ρ_U^∞ represent instead the errors bound at the steady state and finally l_U and l_L can be used to tune the error speed of convergence.

4.2 Control design

Firstly we will need to define the errors associated with the object's position $e_p = [e_{p_1}, e_{p_2}, e_{p_3}]^T : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^3$ and orientation $e_\phi = [e_{\phi_\gamma}, e_{\phi_\theta}, e_{\phi_\psi}]^T : \mathbb{R}_{\geq 0} \rightarrow \mathbb{T}^3$ as:

$$e_p(t) = p_O(q(t)) - p_{O,d}(t) \quad (4.2a)$$

$$e_\phi(t) = \phi_O(q(t)) - \phi_{O,d}(t). \quad (4.2b)$$

Remark 4.2.1. Note that, as stated in 2.2, if $\phi_{O,d}(t) = [\gamma_{o,d}, \theta_{o,d}, \psi_{o,d}]^T$, we will need to have $\psi_{o,d} \neq \pm\frac{\pi}{2}$ in order to avoid singularity positions for $J_{o_r}(x_o)$.

We will require an exponential convergence for the errors, hence we will employ the following performance functions:

$$-\rho_{s_k}(t) < e_{p_k}(t) < \rho_{s_k}(t), \forall k \in \{1, 2, 3\}, \quad (4.3a)$$

$$-\rho_{\phi_l}(t) < e_{\phi_l}(t) < \rho_{\phi_l}(t), \forall l \in \{\gamma, \theta, \psi\}, \quad (4.3b)$$

where $\rho_{s_k}, \rho_{\phi_l} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{>0}$, with:

$$\rho_{s_k}(t) = (\rho_{s_k}^0 - \rho_{s_k}^\infty) e^{-l_{s_k} t} + \rho_{s_k}^\infty, \forall k \in \{1, 2, 3\}, \quad (4.4a)$$

$$\rho_{\phi_l}(t) = (\rho_{\phi_l}^0 - \rho_{\phi_l}^\infty) e^{-l_{\phi_l} t} + \rho_{\phi_l}^\infty, \forall l \in \{\gamma, \theta, \psi\}, \quad (4.4b)$$

are designer-specified, smooth, bounded, positive and decreasing functions of time with $l_{p_k}, l_{\phi_l}, \rho_{p_k}^\infty, \rho_{\phi_l}^\infty, k \in \{1, 2, 3\}, l \in \{\gamma, \theta, \psi\}$ positive parameters incorporating the desired transient and steady state performance respectively.

Next we will define the stacked position error $e_s : \mathbb{R}_{\geq 0} \rightarrow \mathbb{M}$:

$$e_s(t) = \begin{bmatrix} e_{p_1}(t) \\ e_{p_2}(t) \\ e_{p_3}(t) \\ e_{\phi_\gamma}(t) \\ e_{\phi_\theta}(t) \\ e_{\phi_\psi}(t) \end{bmatrix} = x_o(q(t)) - x_d(t), \quad (4.5)$$

and the performance functions matrix $\rho_s : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{6 \times 6}$:

$$\rho_s(t) = \text{diag}\{\rho_{s_1}(t), \rho_{s_2}(t), \rho_{s_3}(t), \rho_{\phi_\gamma}(t), \rho_{\phi_\theta}(t), \rho_{\phi_\psi}(t)\}. \quad (4.6)$$

We will define now the normalized error $\xi_s : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$:

$$\xi_s(t) = \begin{bmatrix} \xi_{s_1}(t) \\ \vdots \\ \xi_{s_6}(t) \end{bmatrix} = \rho_s^{-1}(t)e_s(t) \quad (4.7)$$

and the two signals $\varepsilon_s : \mathbb{R}^6 \rightarrow \mathbb{R}^6$ and $r_s : \mathbb{R}^6 \rightarrow \mathbb{R}^{6 \times 6}$ as:

$$\varepsilon_s(\xi_s) = \begin{bmatrix} \varepsilon_{s_1}(t) \\ \vdots \\ \varepsilon_{s_6}(t) \end{bmatrix} = \begin{bmatrix} \ln\left(\frac{1+\xi_{s_1}}{1-\xi_{s_1}}\right) \\ \vdots \\ \ln\left(\frac{1+\xi_{s_6}}{1-\xi_{s_6}}\right) \end{bmatrix} \quad (4.8)$$

$$r_s(\xi_s) = \text{diag} \left\{ \left[\frac{\partial \varepsilon_{s_m}(\xi_{s_m})}{\partial \xi_{s_m}} \right]_{m \in \{1, \dots, 6\}} \right\} = \text{diag} \left\{ \left[\frac{2}{(1 - \xi_{s_m}^2)} \right]_{m \in \{1, \dots, 6\}} \right\}. \quad (4.9)$$

We can now compute the reference velocity vector $v_{o,des} : \mathbb{R}^6 \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$:

$$\begin{aligned} v_{o,des}(\xi_s, t) &= \begin{bmatrix} \dot{p}_{o,des}(\xi_s, t) \\ \omega_{o,des}(\xi_s, t) \end{bmatrix} = -g_s J_{o_r}(x_o) \rho_s^{-1}(t) r_s(\xi_s) \varepsilon_s(\xi_s) \\ &= -g_s J_{o_r}(\rho_s(t) \xi_s + x_d(t)) \rho_s^{-1}(t) r_s(\xi_s) \varepsilon_s(\xi_s), \end{aligned} \quad (4.10)$$

4.3. Control parameters design

where $g_s \in \mathbb{R}_{\geq 0}$ is a constant gain. We define now the velocity error $e_v : \mathbb{R}^6 \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$ as:

$$e_v(\xi_s, t) = \begin{bmatrix} e_{v_1}(\xi_s, t) \\ \vdots \\ e_{v_6}(\xi_s, t) \end{bmatrix} = v_o(t) - v_{o,des}(\xi_s, t). \quad (4.11)$$

Similarly as what we've seen for the position and orientation error we will proceed now to define the performance functions matrix $\rho_v : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{6 \times 6}$, $\rho_v(t) = \text{diag}\{\rho_{v_m}(t)\}_{m \in \{1, \dots, 6\}}$ where $\rho_{v_m}(t) = (\rho_{v_m}^0 - \rho_{v_m}^\infty) e^{-l_m t} + \rho_{v_m}^\infty$ such that $\rho_{v_m}^0 > \|e_{v_m}(t)\|$, $l_m > 0$ and $\rho_{v_m}^0 > \rho_{v_m}^\infty > 0$, $\forall m \in \{1, \dots, 6\}$.

The normalized velocity error $\xi_v : \mathbb{R}^6 \times \mathbb{R}_{\geq 0}$:

$$\xi_v(\xi_s, t) = \begin{bmatrix} \xi_{v_1}(\xi_s, t) \\ \vdots \\ \xi_{v_6}(\xi_s, t) \end{bmatrix} = \rho_v^{-1}(t) e_v(\xi_s, t), \quad (4.12)$$

and the signals $\varepsilon_v : \mathbb{R}^6 \rightarrow \mathbb{R}^6$ and $r_v : \mathbb{R}^6 \rightarrow \mathbb{R}^{6 \times 6}$:

$$\varepsilon_v(\xi_v) = \begin{bmatrix} \varepsilon_{v_1}(\xi_{v_1}) \\ \vdots \\ \varepsilon_{v_6}(\xi_{v_6}) \end{bmatrix} = \begin{bmatrix} \ln\left(\frac{1+\xi_{v_1}}{1-\xi_{v_1}}\right) \\ \vdots \\ \ln\left(\frac{1+\xi_{v_6}}{1-\xi_{v_6}}\right) \end{bmatrix} \quad (4.13)$$

$$r_v(\xi_v) = \text{diag} \left\{ \left[\frac{\partial \varepsilon_{v_m}(\xi_{v_m})}{\partial \xi_{v_m}} \right]_{m \in \{1, \dots, 6\}} \right\} = \text{diag} \left\{ \left[\frac{2}{(1 - \xi_{v_m}^2)} \right]_{m \in \{1, \dots, 6\}} \right\}. \quad (4.14)$$

We are finally ready to design the distributed control input for each agent $i \in \mathcal{N}$ as $u_i : \mathbb{R}^6 \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^6$:

$$u_i(\xi_v, t) = -c_i g_v J_{o_i}^{-1}(q(t)) \rho_v^{-1}(t) r_v(\xi_v) \varepsilon_v(\xi_v), \quad (4.15)$$

and it's stacked version:

$$\bar{u}(\xi_v, t) = \begin{bmatrix} u_1(\xi_v, t) \\ \vdots \\ u_N(\xi_v, t) \end{bmatrix} = -G^*(q) C_g \rho_v^{-1} r_v(\xi_v) \varepsilon_v(\xi_v) \quad (4.16)$$

where $C_g = \text{diag}\{[c_i g_v]_{i \in N}\} \in \mathbb{R}^{6N \times 6N}$ and c_i are the predefined load sharing coefficients satisfying $\sum_{i \in \mathcal{N}} c_i = 1$ and $0 \leq c_i \leq 1$.

4.3 Control parameters design

We want g_v such that $\|u_i(t)\| \leq \bar{u}$, $\forall t \in \mathbb{R}_{\geq 0}$ where $\bar{u} \in \mathbb{R}_{\geq 0}$ is the designed maximum magnitude for the input. This procedure can be very useful in real scenarios when the torque that each agent's joint can produce is limited. As already stated in 2.2 the torque can be related to the input forces through the kineto-static duality (2.10), so it's possible to design \bar{u} such that $\bar{u} < \|J_i^T(q)\bar{\tau}_i\|$, where $\bar{\tau}_i \in \mathbb{R}^{n_i}$ is the maximum selected torque vector for agent i .

The input forces are computed as in (4.15) thus we need to ensure:

$$\|u_i(\xi_v, t)\| = \|c_i g_v J_{o_i}^{-1}(q(t)) \rho_v^{-1}(t) r_v(\xi_v) \varepsilon_v(\xi_v)\| \leq \bar{u}, \quad (4.17)$$

employng the properties of the matrix norm we will have:

$$\begin{aligned} \|u_i(\xi_v, t)\| &= \|c_i g_v J_{o_i}^{-1}(q(t)) \rho_v^{-1}(t) r_v(\xi_v) \varepsilon_v(\xi_v)\| \\ &\leq c_i g_v \|J_{o_i}^{-1}(q(t))\| \|\rho_v^{-1}(t)\| \|r_v(\xi_v)\| \|\varepsilon_v(\xi_v)\| \\ &\leq c_i g_v \bar{J}_{o_i}^{-1} \frac{1}{\bar{\rho}_v^\infty} \bar{r}_v \bar{\varepsilon}_v, \end{aligned}$$

where $\bar{J}_{o_i}^{-1} = \left(1 + \max_{i \in \mathcal{N}} \{ \|p_{O/E_i}^{E_i}\| \}\right) \leq \|J_{o_i}^{-1}(q(t))\|$, $\bar{\rho}_v^\infty = \min_{m \in \{1 \dots 6\}} \{\rho_{v_m}^\infty\}$, $\bar{r}_v = \max_{m \in \{1 \dots 6\}} \{r_{v_m}(\xi_{v_m})\}$.

From section 4.2 we know that:

$$\|\varepsilon_v(\xi_v(\xi_s(t), t))\| \leq \bar{\varepsilon}_v = \max \left\{ \|\varepsilon_v(\xi_v(\xi_s(0), 0))\|, \frac{\bar{B}_v \max_{m \in \{1 \dots 6\}} \{\rho_{v_m}^0\}}{\lambda_{\min}(\bar{M}^{-1}) g_v} \right\}$$

where \bar{B}_v is a positive constant term satisfying:

$$\bar{B}_v \geq \left\| \frac{1}{m} [\bar{C} v_O - \bar{g} - \bar{w}] - \dot{v}_d - \dot{\rho}_v \xi_v \right\|$$

where m is such that $m \leq \|\bar{M}\|$.

$$1) \bar{\varepsilon}_v = \frac{\bar{B}_v \max_{m \in \{1 \dots 6\}} \{\rho_{v_m}^0\}}{\lambda_{\min}(\bar{M}^{-1}) g_v} :$$

If that's the case we will have to guarantee:

$$c_i \bar{J}_{o_i}^{-1} \frac{1}{\bar{\rho}_v^\infty} \bar{r}_v \frac{\bar{B}_v \max_{m \in \{1 \dots 6\}} \{\rho_{v_m}^0\}}{\lambda_{\min}(\bar{M}^{-1})} \leq \bar{u} \quad (4.18)$$

which does not depend on g_v holding that we need to choose \bar{u} high enough to satisfy this inequality, which make sence since we will need at least a minimum input just to overcome gravity.

$$2) \bar{\varepsilon}_v = \|\varepsilon_v(\xi_v(\xi_s(0), 0))\| :$$

In this case we need to guarantee:

$$c_i g_v \bar{J}_{o_i}^{-1} \|\bar{r}_v \varepsilon_v(\xi_v(\xi_s(0), 0))\| \leq \bar{u},$$

and if we solve it for g_v we will have:

$$g_v \leq \frac{\bar{u} \bar{\rho}_v^\infty}{c_i \bar{J}_{o_i}^{-1} \bar{r}_v \|\varepsilon_v(\xi_v(\xi_s(0), 0))\|}. \quad (4.19)$$

Chapter 5

Hardware

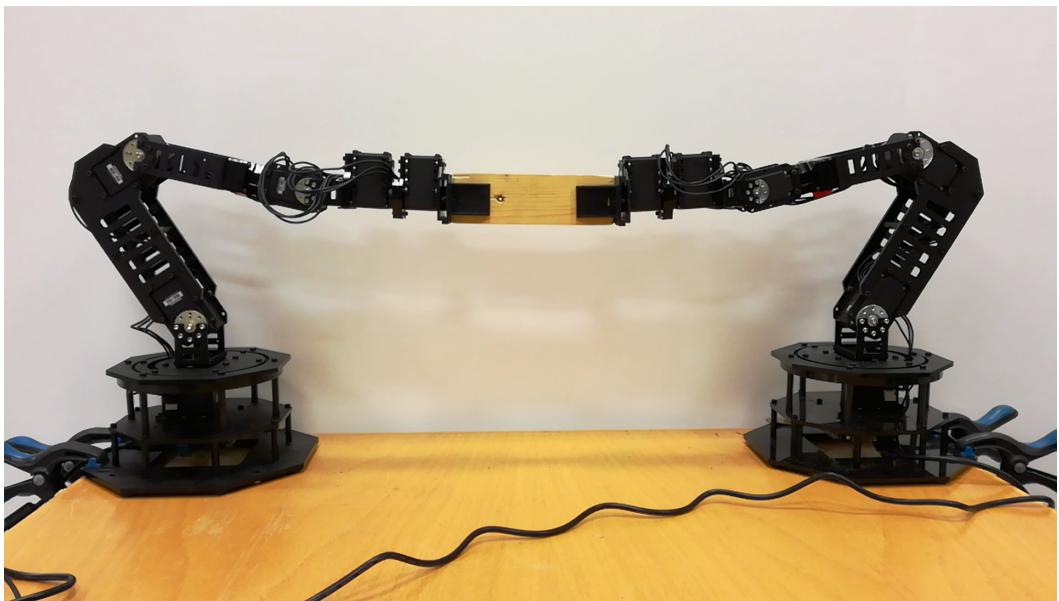


Figure 5.1: Two WidowX mark II arms grasping an object.

In order to perform experiments and simulations we will need to use two WidowX mark II arms grasping a wood rectangular cuboid ($14.5\text{cm} \times 4.4\text{cm} \times 2\text{cm}$, weight: 62g), figure 5.1. The WidowX arm has 5 degrees of freedom but, due to the scene configuration, we will be using just 3, resulting in the case of planar cooperative manipulation.

5.1 WidowX mark II

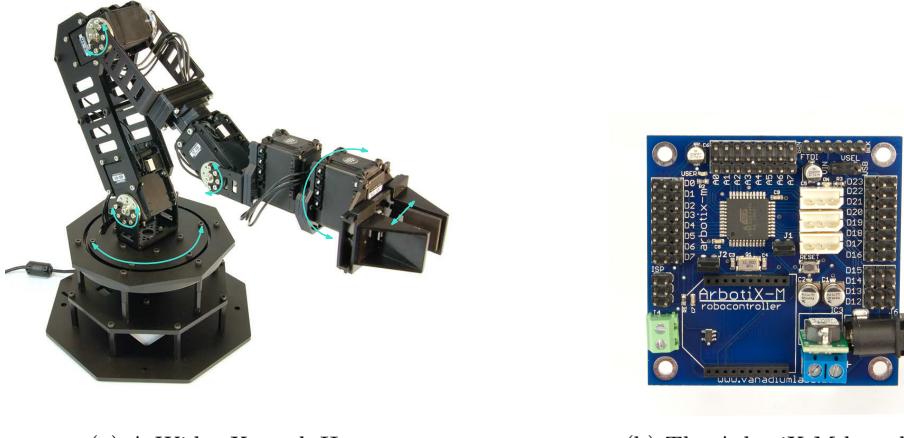
The WidowX mark II robotic arm (figure 5.2a) is a 5 d.o.f. robot manipulator with a parallel pincher produced by Trossen Robotics [21]. The arm can lift up to 400g at 30cm from the base and the gripper is able to hold up to 500g . It's composed of 6 servos in total produced by Robotis [22]:

- 2 Dynamixel MX-28T, figure 5.3b: the first is used to rotate the base link and the

latter as 4th joint. They provide a maximum torque of $2.5Nm$, and a 4096 steps position encoder.

- 2 Dynamixel MX-64T, figure 5.3c: this servos are used in the second and third joint of the arm. They provide a maximum torque of $6Nm$, and a 4096 steps position encoder.
- 2 Dynamixel AX-12A, figure 5.3a: used to open and close the gripper and in the last joint. They provide a maximum torque of $1.5Nm$, and a 1023 steps position encoder.

Like anticipated we will be working with just 4 of this servos, the 2 MX-64T (second and third joints) with the MX-28T (fourth joint) will provide the 3 d.o.f. needed for the experiments, and the AX-12A will be used to control the gripper.



(a) A WidowX mark II arm.

(b) The ArbotiX-M board.

Figure 5.2: The WidowX robotic arm and its ArbotiX-M control board.



(a) Dynamixel AX-12A.

(b) Dynamixel MX-28T.

(c) Dynamixel MX-64T.

Figure 5.3: Arm's servos.

The servos come with an internal velocity and position PID controller. To send and retrieve information to the servo it's sufficient to access the data stored in it's internal register. To read and write on the register we will use an ArbotiX-M board (figure 5.2b), the producer offers an useful .ino script that allow us to easily communicate with the servos using python. Some useful entries of the register are:

- Current position and velocity of the rotor.

- Goal position and velocity for the internal PID controllers.
- PID controller constants.
- Maximum current in terms of % w.r.t. the flowing current when the servo is stalled at 12V.
- The motor sequential ID that allow us to identify the specific servo connected to the board.
- A fixed delay applied before answering a request.

5.2 Servos Characteristics

The servos are not all capable to automatically set a given torque value. Instead it's possible to set a maximum value for the flowing current, hence we need to compute the current-torque relation for each of the MX servos. From the producer page it's possible to download the servos datasheets, there is no explicit formula to compute the torques but an useful table with current-torque values is given. In order to compute an usable relation we performed, in Matlab, an LS estimation on the given data using a linear formulation for the Dynamixel MX-64 and a quadratic formulation for the MX-28 (*servos_characteristics.m*). The estimation returned the following results (Figure 5.4):

$$I_{MX28} = 0.2258\tau^2 + 0.4850\tau + 0.1514 \quad (5.1)$$

$$I_{MX64} = 0.8606\tau + 0.1047 \quad (5.2)$$

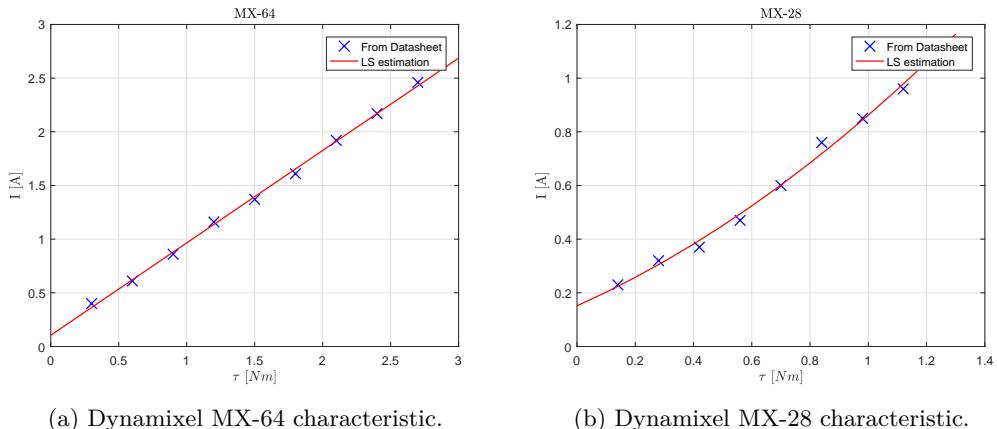


Figure 5.4

Chapter 6

Simulation Environment

All simulations have been carried out using V-REP a virtual experimentation platform developed by Coppelia Robotics that offers some very useful APIs to interface the simulated robots with the ROS framework [23].

In order to perform any kind of simulation a 3D model that allows the software to compute the physics of the scene is needed. As presented in chapter 5 we are going to use two WidowX Mark II robotic arms for our experiments. In the official V-REP repository there is no 3D model for the arm we were going to use, fortunately the producer offers a 3D model compatible with STEP and Adobe Inventors formats (figure 6.1).



Figure 6.1: WidowX 3D STEP model.

Starting from the available model and following the Copellia Robotics guide [24] the scene in figure 6.2 have been created. In order to stream the joints variables and retrieve the torques values from the ROS frameworks a child script have been attached to each robot. The script is composed by four parts:

1. Initialization: In this section all ROS topics and message variables are defined and initialized. Joints position will be published in the `'/robot_name/joints_poses'` topic

and joints velocityes will be published in the '`/robot_name/joints_vels`' topic. Torques will be retrieved from the '`/robot_name/torques`' topic.

2. Sensing: this section contains all the code needed to read and stream joints positions and velocities.
3. Torque Callback: it's actually a function and it's called each time the controller publishes a torque message in the topic. It takes the published values and apply the desired torques at the joints. Similarly to the experimental case there is no straightforward method to set a target torque in V-REP but it's possible to limit the joint maximum torques and set a speed goal. It's then sufficient to limit the torque at the desired values and make the joint rotate with a very high speed in the desired direction.
4. Cleanup: this section is executed each time the simulation is stopped and it just delete all the created topics.

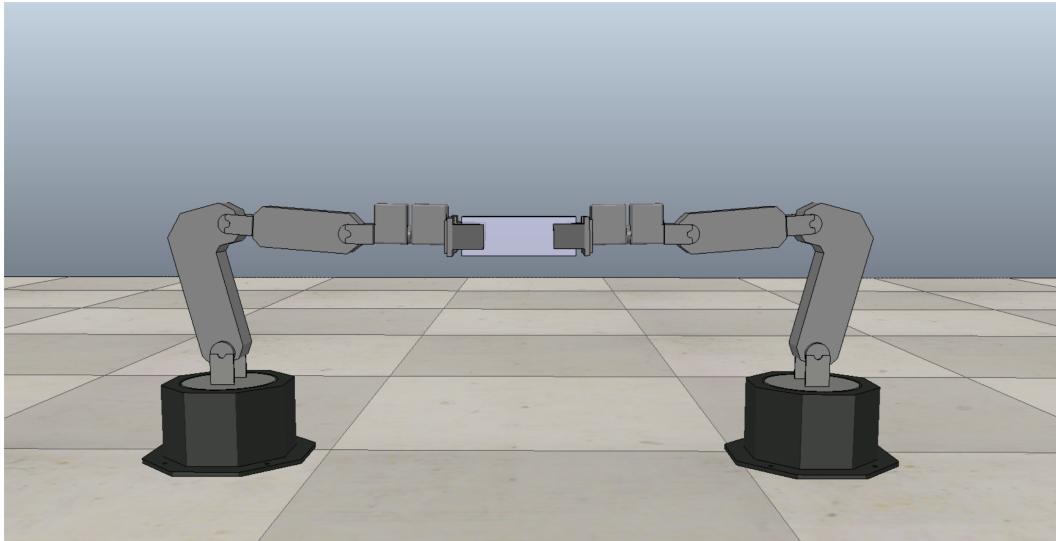


Figure 6.2: Two WidowX mark II arms grasping an object in V-REP.

Chapter 7

Code

In this chapter we will present the code created in order to compute the dynamic matrices, move the robots and implement the controllers.

7.1 Dynamic Matrices

The dynamic matrices computation is divided in three parts:

1. First the *three_link_arm_dynamics.m* script computes the matrices for a generic three link arm using only symbolic values.
2. Then the *evaluate_matrices_3links_state_space.m* script substitutes all the generic parameters in the matrices with the WidowX values, leaving only the joint variables, finally it computes the Jacobian matrix determinant to determine its singularity configurations. Since it's very simple a more in depth explanation is not needed.
3. Finally the computed matrices are copied in the *widowx_compute_dynamics.py* script which implements a class that allows to easily compute the matrices values taking the joints variables as input.

three_link_arm_dynamics.m

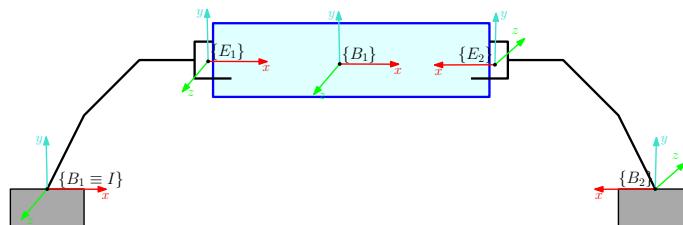


Figure 7.1: Bases (B_i), end effectors (E_i), object (O) and reference (I) frames.

As previously stated this script computes the dynamic matrices for a generic three link arm. All matrices have been computed according to the Lagrangian formulation using the Denavit-Hartenberg convention to place the joints and links frames [18], the inertial frame coincides with the base frame of the first arm, a schematic representation of the most important frames is displayed in figure 7.1, as we can see the second robot is rotated of 180°

along the y axis and translated along the x axis.

The script firstly defines all the needed symbolic variables:

- the links' length, center of mass, weight and inertias;
- the joints' position, velocity, gear reduction, weight and inertias;
- the second robot base offset w.r.t. the reference frame.

Then using the *rotation_matrix.m* and *translation_matrix.m* scripts we compute all the transformation matrices between the links center of mass and motors.

From the transformation matrices just computed we will extract the position vectors of links and joints as the first three elements of the last column of each matrix and the rotation matrices as the first 3×3 diagonal block. By differentiating the position vectors w.r.t. the joint variables we compute the links ($J_L^{(l_i)}$) and joints ($J_L^{(m_i)}$) linear jacobians. Since we only have revolute joints the angular jacobian computation is straightforward:

$$J_A^{l_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, J_A^{l_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}, J_A^{l_3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$J_A^{m_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ k_{r1} & 0 & 0 \end{bmatrix}, J_A^{m_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & k_{r2} & 0 \end{bmatrix}, J_A^{m_3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & k_{r3} \end{bmatrix}$$

We can now compute the joint-space dynamical matrices as:

$$B(q) = \sum_{i=1}^3 \left(m_{l_i} J_L^{(l_i)T} J_L^{(l_i)} + J_A^{(l_i)T} R_i I_{l_i}^i R_i^T J_A^{(l_i)T} + m_{m_i} J_L^{(m_i)T} J_L^{(m_i)} + J_A^{(m_i)T} R_{m_i} I_{m_i}^i R_{m_i}^T J_A^{(m_i)T} \right);$$

$$N_{(i,j)}(q, \dot{q}) = \sum_{k=1}^3 \frac{1}{2} \left(\frac{\partial B_{(i,j)}(q)}{\partial q_k} + \frac{\partial B_{(i,k)}(q)}{\partial q_j} + \frac{\partial B_{(j,k)}(q)}{\partial q_i} \right) \dot{q}_k;$$

$$p(q) = \begin{bmatrix} \frac{\partial U}{\partial q_1} & \frac{\partial U}{\partial q_2} & \frac{\partial U}{\partial q_3} \end{bmatrix}^T.$$

Where U is the potential energy in the Lagrangian equation. Then, in order to simplify computations, all variables known to be zero are set as such and the state-space matrices (M , C and g) are computed as in (2.11).

Now that we have the complete state space dynamical model we want to compute its regression matrices as described in section 3.3. The (3.17) equation is computed and through

a simple inspection the following dynamic parameters are extracted:

$$\theta_i = \begin{bmatrix} l_{c_x,1}^2 m_1 \\ l_{c_x,1} m_1 \\ l_{c_y,1}^2 m_1 \\ l_{c_y,1} m_1 \\ I_{c_1} \\ I_{r_1} \\ l_{c_x,2}^2 m_2 \\ l_{c_x,2} m_2 \\ l_{c_y,2}^2 m_2 \\ l_{c_y,2} m_2 \\ m_2 \\ m_{r_2} \\ I_{c_2} \\ I_{r_2} \\ l_{c_x,3}^2 m_3 \\ l_{c_x,3} m_3 \\ l_{c_y,3}^2 m_3 \\ l_{c_y,3} * m_3 \\ m_3 \\ m_{r_3} \\ I_{c_3} \\ I_{r_3} \end{bmatrix} \quad (7.1)$$

where $I_{c_i} \in \mathbb{R}_{\geq 0}$ and $I_{r_i} \in \mathbb{R}_{\geq 0}$ are the links and motors inertias values, $m_i \in \mathbb{R}_{\geq 0}$ and $m_{r_i} \in \mathbb{R}_{\geq 0}$ are its masses, $l_{c_x,i} \in \mathbb{R}_{\geq 0}$ and $l_{c_y,i} \in \mathbb{R}_{\geq 0}$ the coordinate of links' center of masses w.r.t. the frame they're attached. Finally the elements in the vector are collected from the equation and stored in the regression matrix.

If we apply the same procedure for the equation (3.16) we will obtain:

$$\theta_O = \begin{bmatrix} m_O \\ I_{z_O} \end{bmatrix}. \quad (7.2)$$

widowx_compute_dynamics.py

After computing in Matlab the M_i , C_i and g_i , $i = 1, 2$, matrices we implemented them in this script creating a class named "*WidowxDynamics*" that allows us to easily include the dynamic computation in the ROS nodes we will be using to control the robots. The class includes the following methods:

- *compute_ee_pos(self, r1_array_poses)*: it takes the joints positions numpy array as input and returns the end effector position as a numpy array for the robot with the inertial frame attached to its base.
- *compute_ee_pos(self, r2_array_poses, ambient)*: it takes the joints positions numpy array and a string "*ambient*" as inputs and returns the end effector position as a numpy array for the robot shifted from the inertial frame. The "*ambient*" parameter can be set to "*sim*" or "*exp*" in order to take in to account of different offsets between experiments and simulations without changing the class. To set the offset variables one can change the "*x_off*" and "*x_off_sim*" parameters in the class declaration.

- `compute_jacobian1(self, r1_array_poses)` and `compute_jacobian2(self, r2_array_poses)`: they take the joints positions numpy array as input and return the end effector jacobian as a numpy matrix for the first and second robot respectively.
- `compute_M1(self, r1_array_poses)` and `compute_M2(r2_array_poses)`: they take the joints positions numpy array as input and return the inertia matrix as a numpy matrix for the first and second robot respectively.
- `compute_C1(self, r1_array_poses, r1_array_vels)` and `compute_C2(self, r2_array_poses, r2_array_vels)`: this methods take the joints positions and velocities numpy arrays as inputs and return the Coriolis matrix as a numpy matrix for the first and second robot respectively.
- `compute_g1(self, r1_array_poses)` and `compute_g2(self, r2_array_poses)`: they take the joints positions numpy array as input and returns the gravity vector as a numpy array for the first and second robot respectively.
- `compute_Mo(self, Ro)`: it takes the object rotation matrix as input and returns its inertia matrix as a numpy matrix.
- `compute_Y1(self, r1_array_poses, r1_array_vels, v_o_r1, v_o_r_dot1)` and `compute_Y2(self, r2_array_poses, r2_array_vels, v_o_r2, v_o_r_dot2)`: they take the joint variables and the velocity and acceleration reference signals numpy arrays as inputs and return the regression matrix for the first and second arm respectively.
- `compute_Yo(self, v_o_r_dot)`: using the reference velocity signal as input returns the object regression matrix.

7.2 Driver

Each motor includes a register where all joints variables and target signal are stored, the communication happens through a serial to USB port, and the needed communication protocol is implemented in the `ros.ino` script, preuploaded by the producer in the *ArbotiX-M* board. To access each register an useful python class, implemented in the *arbotix_python* package (`src/arbotix.py`), is provided by the producer [21].

From the beginning we encountered some problems with the registers access speed. If we tried to read all joint variables or write all the torque goals the communication frequency were limited at 30Hz. In order to speed up readings and writings some additional methods have been implemented: `syncSetTorque(torques, position)`, `syncGetPos(self, IDs)` and `syncGetVel(self, IDs)`. Those methods allow us to read and write on multiple motors at the same time, in this way we can achieve n times the standard update frequency where n is the number of servos we are using. In our case with the standard version we could achieve a transfer rate of 30Hz for the readings of all 6 joint variables (12 bytes), with the new methods we increased that value up to 180Hz. Finally by decreasing the delay time that each servo takes to answer a register call we have been able to increase the reading frequency of about 20Hz, reaching 200Hz, a more than suitable value for our purpose.

The values returned by the *ArbotiX* class are unsigned integers of 1 or 2 bytes length. All the conversions are reported in the servos datasheets [22] and the useful ones have been reported in the `servos_parameters.py` file for an easy access in the driver. We now have some specifications we want to meet with our software:

- It has to limit the maximum applicable joints torques: the MX-64 and MX-28 have a stall torque of $6Nm$ and $2.5Nm$ respectively, in order to not overheat the servos we limited that value at 50% of its maximum.
- It has to check for jacobian singularity positions that may result in unstableness for the controllers.
- It needs to implement some functions that allow us to easily shut everything down, setting all torques to zero.
- It has to continuously publish in a topic the joints variables with a refresh rate of at least 60Hz.
- Finally it needs to be able to listen and apply the torques published from a controller in a topic. As stated in section 5.2, the servos are not all capable to automatically set a given torque value. Instead it's possible to set a maximum value for the current circulating in the servo, since we have the torque-current relation, it's possible to setup a simple workaround. From the desired torque we will compute the current needed to achieve our goal and use this value as upper limit in the servo, then the only thing we need to do is send a position message to the servo in order to make it move according to the desired direction and if the goal is far enough from the current position it will move using all the available torque/current.

The class implementing all this functions is contained in the `widowx_state_space_driver.py` file of the `widowx_driver` package, and the driver can be launched using the `coop_driver.launch` file with a roslaunch command.

`widowx_state_space_driver.py`

The driver is implemented trough the `WidowxNode` class and inherits the `ArbotiX` class. It contains the following methods:

- `__init__(self, serial_port, robot_name)`: the initialization method takes two strings as input, the robot's name (`robot_name`) and the serial port the robot is connected to (`serial_port`, usually `/dev/ttyUSBK` where K is a sequential positive integer). Firstly the method sets up the connection with the `ArbotiX-M` board using the `__init__` method of the `ArbotiX` class giving the serial port name as input, after that it waits 10 seconds in order to allow all the servos to connect. Once all motors are connected it resets its *maximum velocity* and *PID* registers in order to be sure that no residual values have remained from previous experiments, then the robot is placed at it's initial position and after 3 second the gripper is closed (Figure 5.1). *Maximum velocity* and *PID* register are then set to the values specified in the launch file and the ROS topics setup starts. The node initializes 2 publishers, 2 listeners and a service:
 - `pos_pub` and `vel_pub` are the publishers for joints positions and velocities respectively, the topics are named '`/widowx_3links_robot_name/joints_poses`' and '`/widowx_3links_robot_name/joints_vels`', `Float32MultiArray` is used as message format;
 - `torque_sub` is the subscriber for the torque messages sent by the controller, the topic needs to be named '`/widowx_3links_robot_name/torques`', `Float32MultiArray` is used as message format;

- *gripper_sub* is the subscriber for the gripper position, *Bool* is used as message format, if *True* the gripper will close, the topic needs to be named '*widowx_3links_robot_name/gripper*';
- *sec_stop_server* is the service handler for the security stop service, if called it will shut down the driver. The service is called '*widowx_3links_robot_name/security_stop*'

Finally it calls the *publish()* method.

- *publish(self)*: in this method it's implemented all the sensing code for the joints variables. It retrieves the joints positions and velocities register values, checks for error in the process, if something went wrong it notifies the user with a message in the terminal and tries again to retrieve the data. Once a good value is received, position and velocity register values are converted in [rad] and [rad/s] respectively. Then it checks if the configuration is of "near singularity" for the WidowX jacobian matrix. Its determinant is $0.0213\cos(q_2 - 1.246)$ so the only reachable Jacobian singularity is in $q_2 = -0.24\text{rad}$. If the second servo position is approaching the singularity the driver notifies the user and if it's nearer than 0.125rad it shuts down the node. Finally joints position and velocities are published.
- *_compute_currents(self, torques, directions)*: the method takes two lists as inputs, the desired torques and the rotating direction of the joint (the direction can be any number different from 0, it just needs to be positive for counterclockwise rotations and negative for clockwise rotations), the values in the two lists are ordered from the first joint to the end effector. The method applies 5.1 and 5.2 to compute the current needed to obtain the desired torque. A corrective gain is added to the constant value in order to take into account internal friction of the motors, to compute its sign the moving direction is needed. This value is not included in the datasheets and has been estimated from the minimum amount of torque needed to make the rotor move. It returns a list containing the computed values ordered as the inputs.
- *_torque_callback(self, msg)*: handles the callbacks from the torques topic. The desired torques are stored in the first three positions of the *msg.data* field, in the positions from 4 to 6 the desired rotating direction is stored, it will be a positive number if counterclockwise, negative otherwise. The method applies a saturation on the torque values and computes the index values corresponding to the currents returned from *_compute_currents(self, torques, directions)*. If saturation occurs a message in the terminal is displayed and finally it applies the torques using the *ArbotiX* class.
- *_gripper_callback(self, msg)*: handles the callbacks from the gripper subscriber. It uses the *ArbotiX* class to move the servo that opens and closes the gripper.
- *_sec_stop(self, req)*: handles the security stop service, it takes as request a string containing an explanation for the forced shut down and simply stops the node.
- *tourn_off_arm(self)*: this method sets all torques to zero stopping the arm. It's called automatically anytime the node is shut down.

coop_driver.launch

In this file is reported an example on how to launch the driver for two robots. It's possible to upload in the ROS parameter server the robot name, the serial port where the arm is connected, the maximum allowed velocity for the joints, the values of the motors internal PID and finally the arm initial positions in terms of the index value of each joint.

7.3 Controllers

In this section we are going to explain the code that implements the controllers designed in chapters 3 and 4. The controllers used for simulations and experiments are coded in the *widowx_controller* ROS package, more precisely the *widowx_cooperative_state_space_sim_3links_controller.py*, *widowx_cooperative_state_space_sim_3links_adaptive.py* and *widowx_PPC_sim.py* scripts contain the code used for simulations. Experiments' controllers instead can be found in the *widowx_cooperative_state_space_3links_controller.py*, *widowx_cooperative_state_space_3links_adaptive.py* and *widowx_PPC.py* scripts. The two implementations are very similar. A difference resides in the control parameters: the real model is, for obvious reasons, different from the simulated one therefore they require ad-hoc calibration of the control constants. Another difference is the presence of a synchronization exploit needed to make the controller wait for V-REP to publish the new joints variables during simulations. All controller are implemented as a python class, can be run as a stand-alone executable and contains the same methods:

- *_init_(self)*: initialize the controller defining all control variables, matrices and signals. It subscribes to the topics to listen to joint variables and desired trajectory and starts up the topic top publish the desired torques message. It creates the *ComputeDynamics* class object and finally initializes the security stop service handler. By leaving this operation at the end we ensure to always have the driver running when the control computation starts, in fact the controller will wait for the driver to startup the security stop service before terminating the initialization.
- *_r1_pose_callback(self, msg)*, *_r2_pose_callback(self, msg)*, *_r1_vel_callback(self, msg)* and *_r2_vel_callback(self, msg)*: are all the callback methods that handles the messages published in the joints velocity and position topics. Once a message is published the method copy its value in a class variable that is then going to be used in the control law.
- *_target_callback(self, msg)*: every time a message is published in the target trajectory topic this method is called. It stores the desired position velocity and acceleration in three different global variables for further use.
- *compute_torques(self)*: is called at the end of the initialization process and cyclically computes the torques that needs to be applied for the given joint variables and desired trajectory following the control laws described in chapters 3 and 4.

Chapter 8

Simulation Results

In this chapter we are going to present the simulations results obtained running the coded controllers in the V-REP environment. The robots are required to cooperatively move the object following a designed trajectory:

$$x_{o,d}(t) = 0.35 + 0.05\sin(\omega t) \quad (8.1a)$$

$$y_{o,d}(t) = 0.15 - 0.05\cos(\omega t) \quad (8.1b)$$

$$\psi_{o,d}(t) = -\frac{\pi}{20}\sin(2\omega t) \quad (8.1c)$$

$$\omega = \frac{2\pi}{T} \quad (8.1d)$$

$$T = 15s \quad (8.1e)$$

the trajectory has been designed to move all joints at the same time, avoiding Jacobian singularity configurations while spacing in all the three degrees of freedom of the robot. In order to make the results comparable the same trajectory have been used for every controller. Finally the following load sharing coefficients have been applied:

$$c_1 = 0.75$$

$$c_2 = 0.25.$$

8.1 Quaternion based approach

As stated in chapter 5 all simulations are going to be performed using just three degree of freedom producing a movement confined in the (x, y) plane with rotations only along the z axis. This allow us to reduce the dimensions of the vectors representing position and orientations and consequentially of all the control signals used to compute the output

torques. We will have:

$$\begin{aligned}
 p_O &= \begin{bmatrix} p_x \\ p_y \end{bmatrix} \in \mathbb{R}^2, \\
 \phi_O &= \psi_O \in \mathbb{T}, \\
 e_p &= \begin{bmatrix} e_x \\ e_y \end{bmatrix} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2, \\
 e_\xi &= \begin{bmatrix} e_\eta \\ e_\varepsilon \end{bmatrix} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{S}, \\
 e_\omega &= \dot{\psi}_O - \dot{\psi}_{O,d} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}, \\
 u_i &: \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^3, i = 1, 2, \\
 K &= \text{diag}\{k_x, k_y, k_\varepsilon\} \in \mathbb{R}_{\geq 0}^{3 \times 3}, \\
 K_v &= \text{diag}\{k_{vx}, k_{vy}, k_{vw}\} \in \mathbb{R}_{\geq 0}^{3 \times 3}.
 \end{aligned}$$

and thus our goal will be to have:

$$\lim_{t \rightarrow \infty} \begin{bmatrix} e_p(t) \\ |e_\eta(t)| \\ e_\epsilon(t) \end{bmatrix} = \begin{bmatrix} 0_{2 \times 1} \\ 1 \\ 0 \end{bmatrix}$$

While for the non-adaptive part we were able to run the simulations at $60Hz$ the same was not possible for the adaptive implementation. Here we need to compute a 3×22 matrix instead of the 3×3 matrices of the dynamical model adding an non indifferent complexity to the program that forced the simulation to run at $20Hz$.

Non Adaptive

In its basic implementation this controller heavily relies on the computed dynamical model. Since no more accurate information was available in order to compute inertias and center of mass of the arm's links they have been approximated to a bar, this obviously introduces some estimation error in the model computation that negatively afflict the experiments. As we can see in figure 8.1 the trajectory of the object presents a marked delay and a loss in amplitude with respect to the target trajectory. This results in relative errors of about the 10% in position and orientation, figures 8.2 and 8.3.

The gain matrices have been set as follow:

$$K = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 100 & 0 \\ 0 & 0 & 30 \end{bmatrix}; K_v = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (8.2)$$

This revealed to be a good compromise in order to minimize the errors while avoiding to introduce instability in the controller. Lower values will, in fact, increases the delay of the trajectory and thus the errors, while higher values leads overshooting and thus shaking in the two arms movements.

8.1. Quaternion based approach

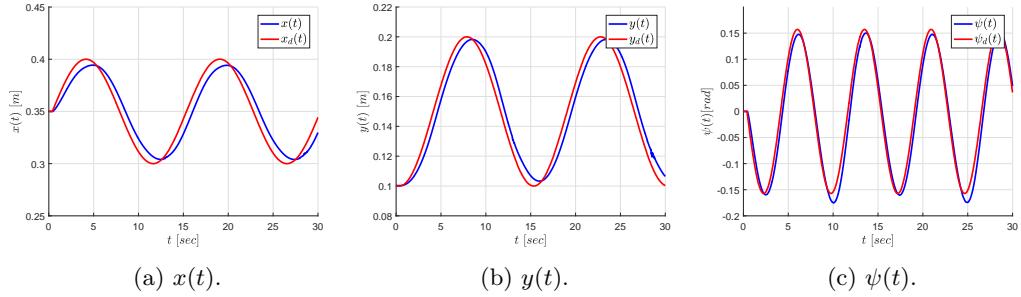


Figure 8.1: Executed and desired trajectories.

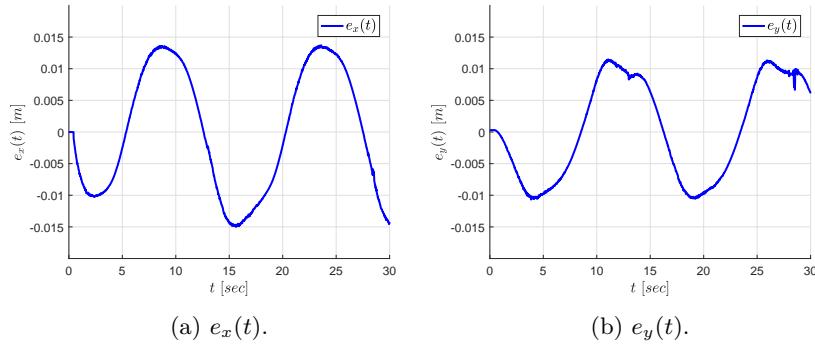


Figure 8.2: Position errors.

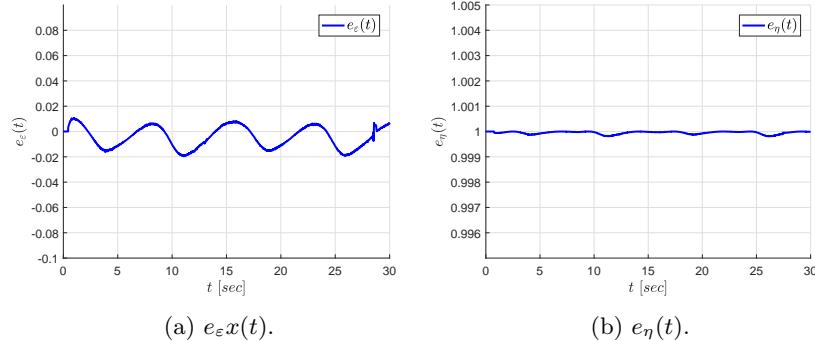


Figure 8.3: Orientation errors.

By looking at the torques in figure 8.4 it's evident the action of the load sharing coefficients that shifts the load on the first arm, notice that the torques are almost always inside the bounds we setted for the two WidowX arms servos, saturation occurs only for very limited periods of time and only marginally affect the simulation results, we can in fact see that when saturation occurs the errors temporarily increase.

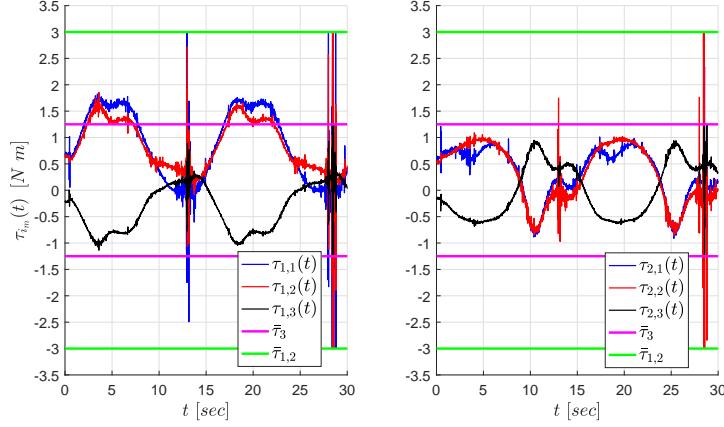


Figure 8.4: Torques inputs for both robots.

Adaptive

As expected by using the adaptive version of the same controller we have been able to significantly reduce both position and orientation errors. We initialized the θ_O and $\theta_i, i = 1, 2$ dynamical parameters vector with the estimated values used in the previous section and then applied the differential equations 3.19 to update the estimation. If we look at the figures 8.5, 8.6 and 8.7 it's possible to notice how at the beginning of the simulation the trend is similar to the non-adaptive case, but then by updating the dynamical parameters vector we are able to halve the errors by almost completely removing the trajectory delay and amplitude error. Moreover we have been able to achieve this results with lower control parameters, increasing the output smoothness and magnitude, figure 8.8, again the load sharing coefficients allowed us to shift most of the work to the first robot.

We used the following control matrices and constants:

$$K = \begin{bmatrix} 15 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 20 \end{bmatrix}; K_v = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \quad (8.3)$$

$$\gamma_1 = \gamma_2 = 0.0005 \quad (8.4)$$

$$\gamma_O = 0.001 \quad (8.5)$$

8.1. Quaternion based approach

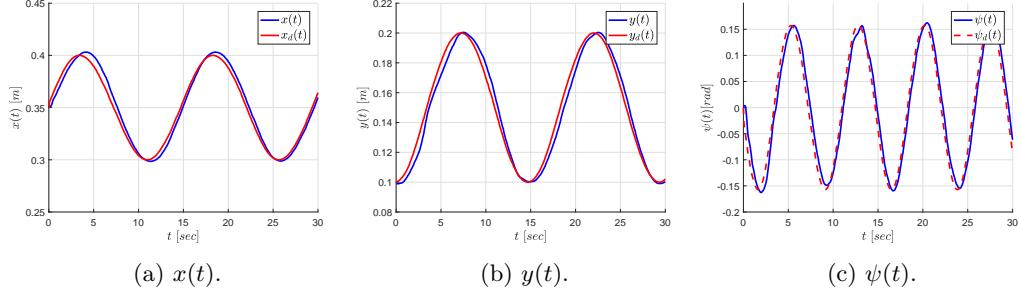


Figure 8.5: Executed and desired trajectories.

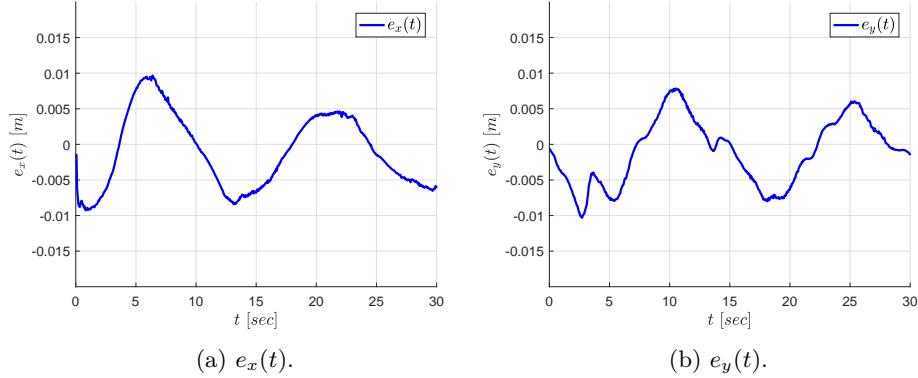


Figure 8.6: Position errors.

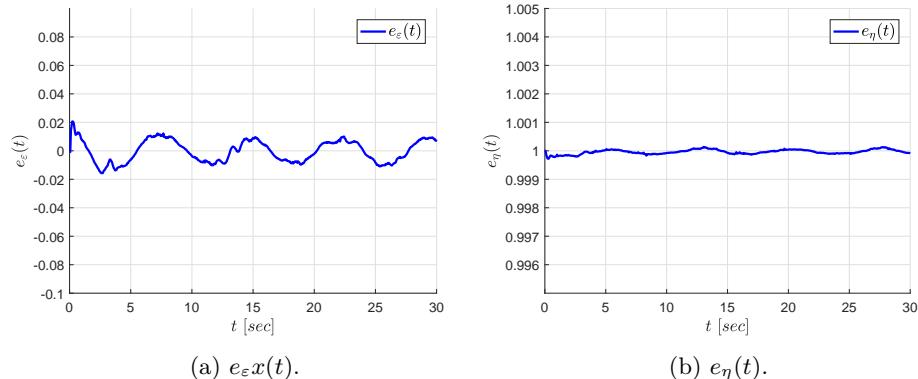


Figure 8.7: Orientation errors.

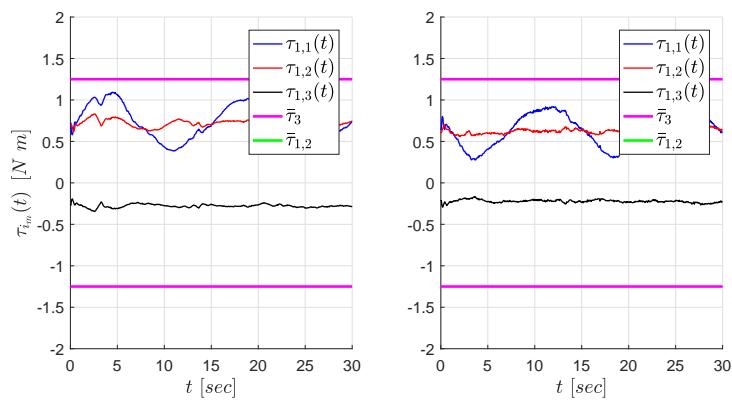


Figure 8.8: Torques inputs for both robots.

8.2 Prescribed Performance Control

As seen in the previous section the quaternion based controller suffers from the approximations in the dynamical parameters of the system, by using the adaptive formulation we can overcome this problem, but it comes with a consistent increment in computational complexity that may cause severe problems if we want to apply the same controller in more complex scenes.

We are now going to present the results obtained running the prescribed performance controller in the V-REP environment, this approach does not rely on the system model, and only the end effector Jacobian computation is needed. This allowed us to keep the update frequency for the simulation at the desired 60Hz .

Similarly to section 8.1 it's possible to introduce some simplifications:

$$\begin{aligned} e_p &= \begin{bmatrix} e_x \\ e_y \end{bmatrix} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2, \\ e_\phi &= e_\psi : \mathbb{R}_{\geq 0} \rightarrow \mathbb{T}, \\ e_s &= \begin{bmatrix} e_p \\ e_\phi \end{bmatrix} = \begin{bmatrix} e_x \\ e_y \\ e_\phi \\ e_\psi \end{bmatrix} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2 \times \mathbb{T}, \\ \rho_k &= \text{diag}\{\rho_{k_x}, \rho_{k_y}, \rho_{k_\psi}\} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{3 \times 3}, \\ \xi_k &: \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^3, \\ \varepsilon_k &: \mathbb{R}^3 \rightarrow \mathbb{R}^3 \\ r_k &: \mathbb{R}^3 \rightarrow \mathbb{R}^{3 \times 3} \\ k &= s, v, \\ u_i &: \mathbb{R}^3 \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^3, i = 1, 2. \end{aligned}$$

We applied the following performance functions:

$$\begin{aligned} \rho_{s_x}(t) &= \rho_{s_y}(t) = (0.04 - 0.01) e^{-0.5t} + 0.01 \\ \rho_\psi(t) &= (0.5 - 0.05) e^{-0.5t} + 0.05 \\ \rho_{v_x}(t) &= (7 - 4) e^{-0.5t} + 4 \\ \rho_{v_y}(t) &= \rho_{v_\psi}(t) = (10 - 5) e^{-0.5t} + 5 \end{aligned}$$

and control parameters:

$$\begin{aligned} g_s &= 0.05 \\ g_v &= 7 \end{aligned}$$

where g_v has been computed employing (4.19) with $\bar{u} = 4$ and thus $\|\tau\| = 4.42\text{Nm} = \|[1.25\text{Nm}, 1.25\text{Nm}, 1.25\text{Nm}]\|$ which is the maximum applicable torque for each of the three joints.

As we can see from figures 8.9, 8.10 and 8.11 after the transient part is over the object follows almost perfectly the desired trajectory, moreover the errors mimics very well the performance functions transient and steady state trend. Finally, except for an initial spike

due to the fact that in the beginning of the simulation the arms tends to fall for gravity, the computed torques never reach the saturation value, figure 8.12. As before it's possible to notice the effect of the load sharing coefficients in the computed torques.

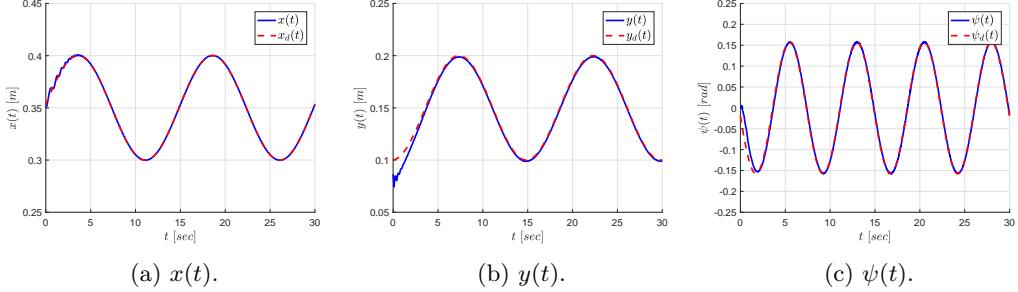


Figure 8.9: Executed and desired trajectories.

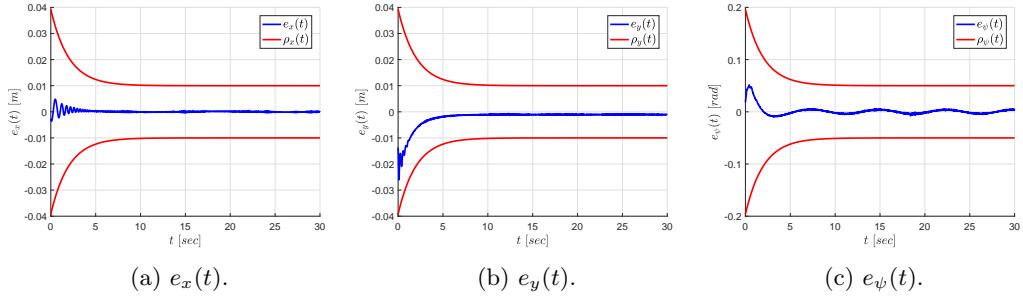


Figure 10: Position errors with associated performance functions.

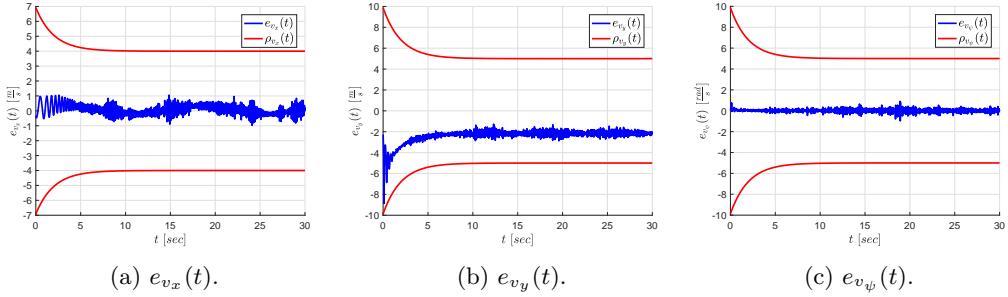


Figure 11: Velocity errors with associated performance functions.

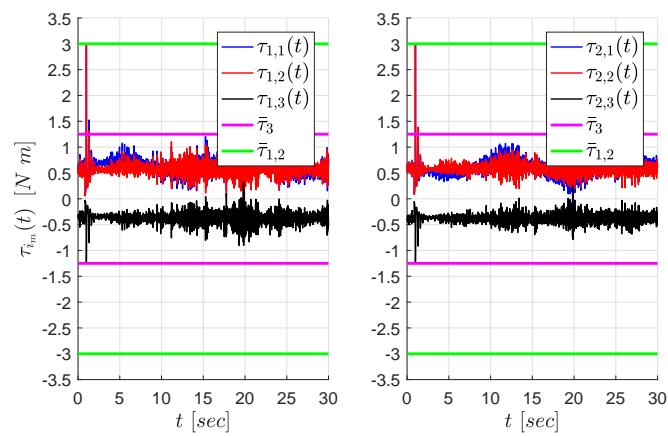


Figure 8.12: Torques inputs for both robots.

Chapter 9

Experimental Results

In this chapter we are going to present the experimental results obtained running the two WidowX Mark II arms with the driver and controllers documented in chapter 7. The object's desired trajectory is as follows:

$$x_{o,d}(t) = 0.301 + 0.05\sin(\omega t) \quad (9.1a)$$

$$y_{o,d}(t) = 0.16 - 0.05\cos(\omega t) \quad (9.1b)$$

$$\psi_{o,d}(t) = -\frac{\pi}{20}\sin(2\omega t) \quad (9.1c)$$

$$\omega = \frac{2\pi}{T} \quad (9.1d)$$

$$T = 35s \quad (9.1e)$$

as before it allows to move all joints at the same time, avoiding Jacobian singularity configurations while spacing in all the three degrees of freedom of the robot. Since not all controller could perform the trajectory with the same period as the one used for simulations we had to raise it and set $T = 35s$. Moreover a longer period will generate lower joints speed and thus lower torques minimizing the risk to damage the servos. As before, to make the results comparable the same trajectory have been used for every controller and finally the following load sharing coefficients have been applied:

$$c_1 = 0.75$$

$$c_2 = 0.25.$$

Note that all the simplifications exposed in the simulations results chapter for position, orientation and control signals vectors and matrices holds for experiments too and will be given for granted. We will see that the trajectory described by the object during experiments is not as smooth as the simulated one, this is due to the difficulty in applying the desired torques at the servos and noise in the joints variable readings. Without a feedback on the applied load it's not possible to verify and adjust the input signal on a driver level. Moreover noise in the joint variable readings results in noise in the computed torques. Future implementations may introduce some low pass filters in order to smooth the joint variables signals removing part of the reading noise.

9.1 Quaternion based approach

As seen while running simulations the adaptive version of the controller is limited by its computational complexity, during experiments we don't have to run V-REP, this results in more available resources that allowed us to work at $30Hz$. For the non adaptive and PPC versions all experiments run at $120Hz$.

Non-Adaptive

The slower trajectory allow us to compensate for imprecisions in the dynamical model. As we can see in the figures 9.1, 9.2 and 9.3 the object follow the desired trajectory with less than 1% of relative error. As before in figure 9.4 we can see the effect of the load sharing coefficients and notice that the torques of each joint never overcome the $1.25Nm$ limit. This results have been obtained with the following control matrices:

$$K = \begin{bmatrix} 50 & 0 & 0 \\ 0 & 50 & 0 \\ 0 & 0 & 80 \end{bmatrix}; K_v = \begin{bmatrix} 3.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}. \quad (9.2)$$

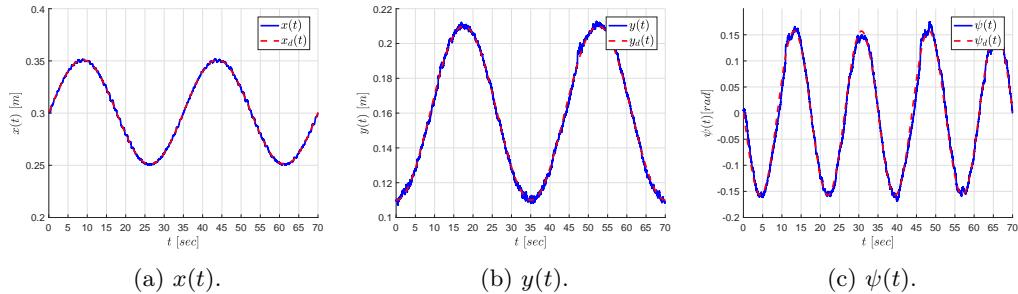


Figure 9.1: Executed and desired trajectories.

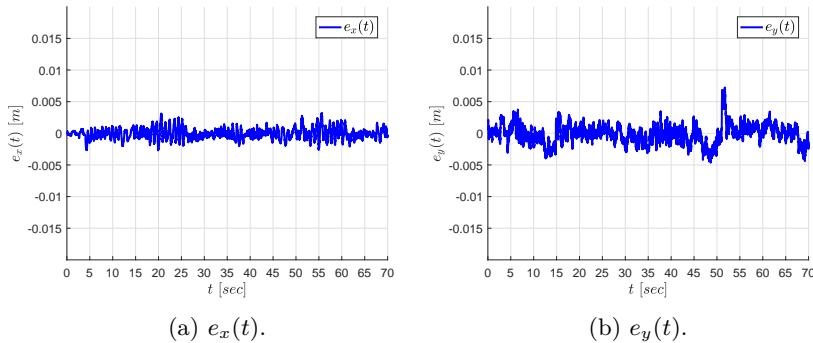


Figure 9.2: Position errors.

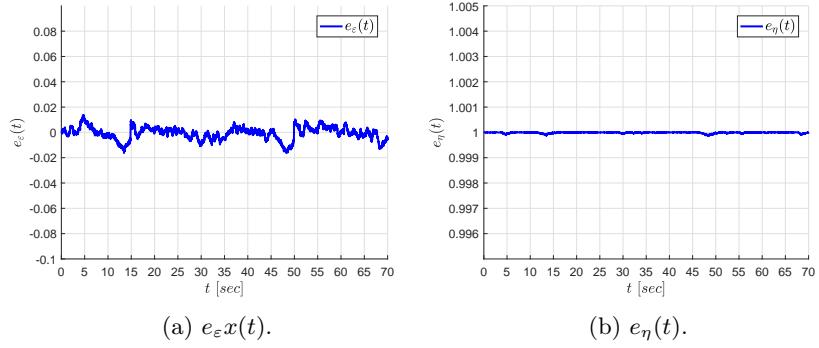


Figure 9.3: Orientation errors.

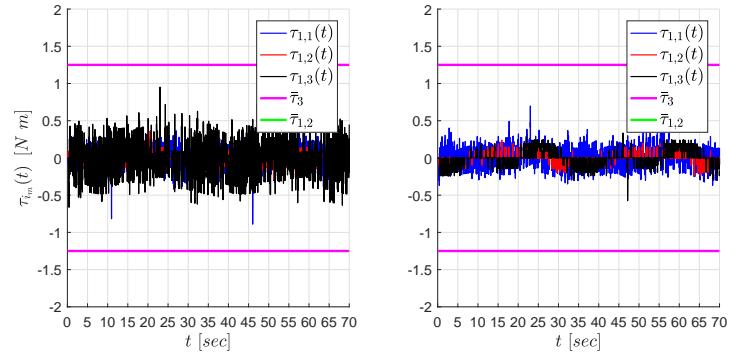


Figure 9.4: Torques inputs for both robots.

Adaptive

During experiments with the adaptive version of the controller, we didn't see the the same expected behavior, while the x trajectory is followed with very low error we can't say the same for the y and ψ ones, figure 9.5. Errors still have low values but they do not diminish with time, figures 9.6 and 9.7. This strange behavior is probably caused by the low updating frequency: the controller can correct deviations from the desired trajectory, but it can't do it fast enough, resulting in occasional overshoots. By looking at figure 9.8, we can again confirm the boundedness of torques and the effect of the load sharing coefficients. Moreover we can see how this approach allow us to compute again a smoother output signal.

These results have been obtained with the following control matrices and constants:

$$K = \begin{bmatrix} 50 & 0 & 0 \\ 0 & 50 & 0 \\ 0 & 0 & 80 \end{bmatrix}; K_v = \begin{bmatrix} 3.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \quad (9.3)$$

$$\gamma_1 = \gamma_2 = 0.0005 \quad (9.4)$$

$$\gamma_O = 0.001 \quad (9.5)$$

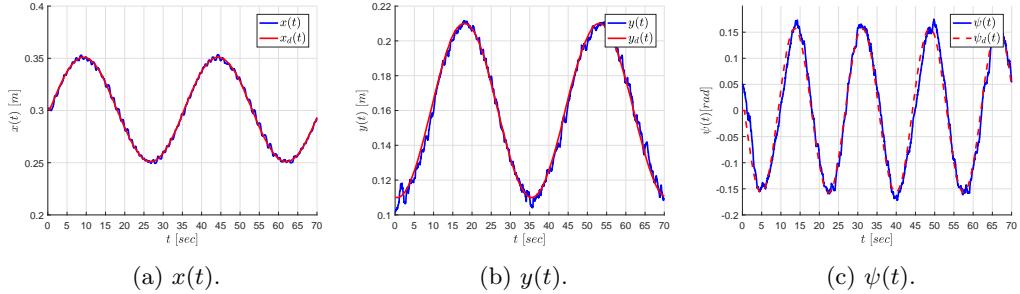


Figure 9.5: Executed and desired trajectories.

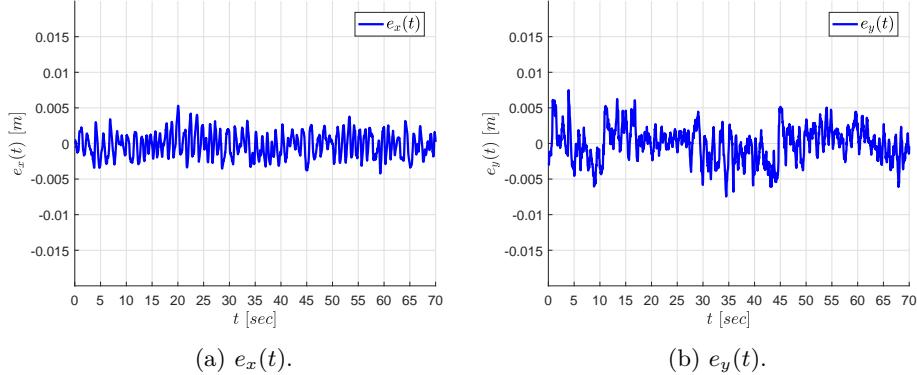


Figure 9.6: Position errors.

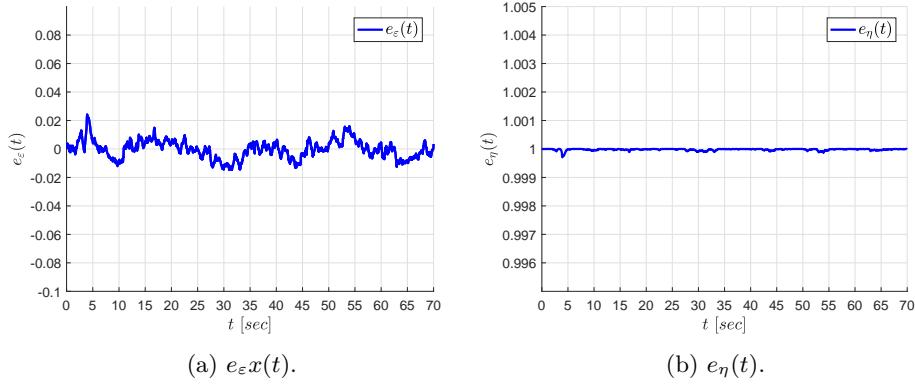


Figure 9.7: Orientation errors.

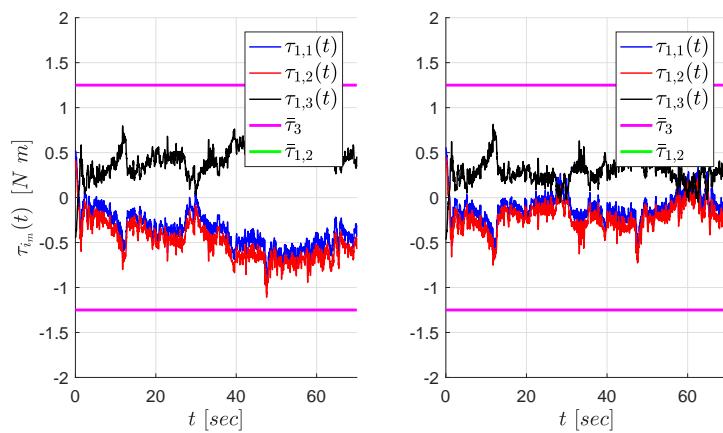


Figure 9.8: Torques inputs for both robots.

9.2 Prescribed Performance Control

The experiments confirmed again the goodness of this approach revealing low tracking errors (figure 9.9) and torques (figure 9.12). By looking at figure 9.10 we can still notice the similarity between position error and performance functions trends. Speed errors does not really follow the performance function, but they're anyway bounded by it, which is all we need in order to ensure the effectiveness of the controller, figure 9.11. Finally, as always by looking at figure 9.12 we can see the boundedness of torques and the effect of load sharing coefficients.

This results have been obtained with the following performance functions and control parameters:

$$\begin{aligned}\rho_{s_x}(t) &= \rho_{s_y}(t) = (0.05 - 0.02)e^{-0.2t} + 0.02 \\ \rho_\psi(t) &= (0.4 - 0.2)e^{-0.2t} + 0.2 \\ \rho_{v_x}(t) &= (10 - 5)e^{-0.2t} + 5 \\ \rho_{v_y}(t) &= (15 - 10)e^{-0.2t} + 10 \\ \rho_{v_\psi}(t) &= (7 - 3)e^{-0.2t} + 3 \\ g_s &= 0.05 \\ g_v &= 6.8\end{aligned}$$

where we decided to take a lower g_v for safety reasons.

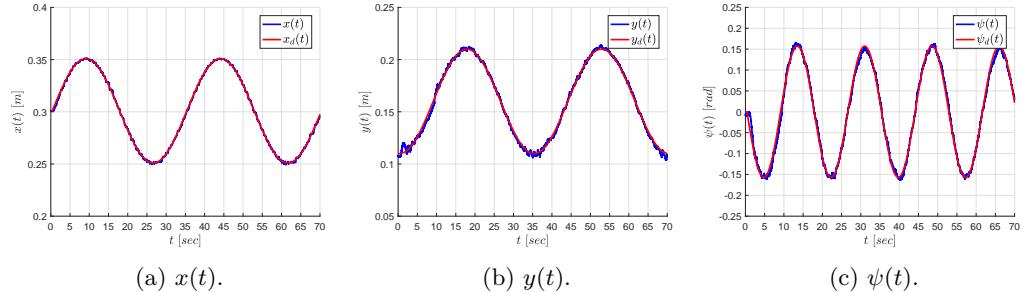


Figure 9.9: Executed and desired trajectories.

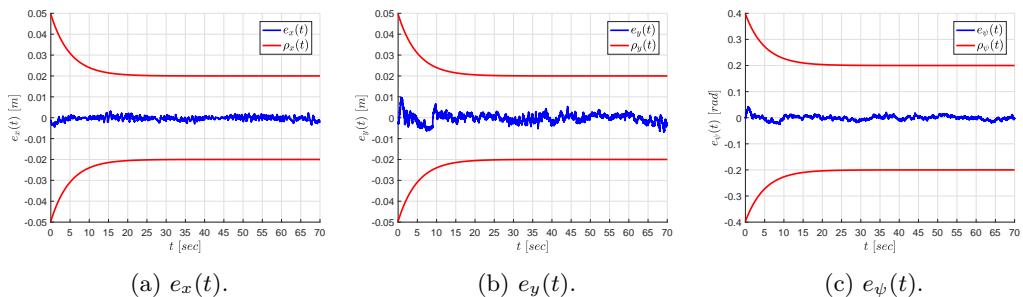


Figure 9.10: Position errors with associated performance functions.

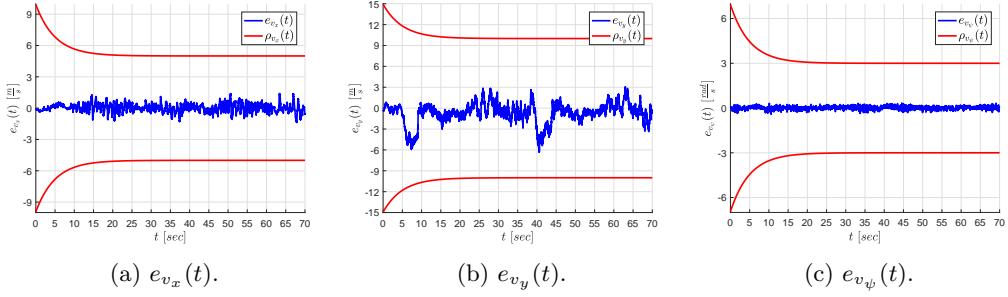


Figure 9.11: Velocity errors with associated performance functions.

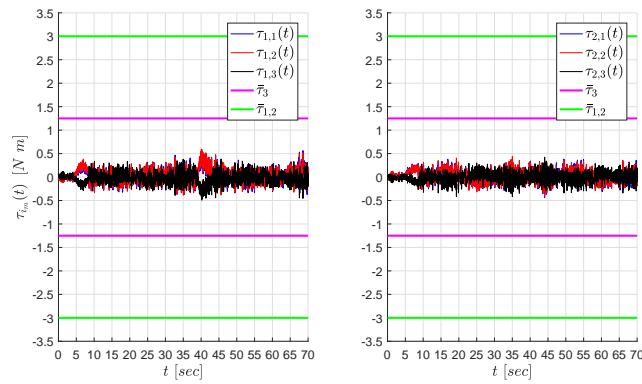


Figure 9.12: Torques inputs for both robots.

Chapter 10

Conclusions

We proposed three novel control protocols for the decentralized cooperative manipulation of an object without employing any force/torque measurements. The first controller employs unit quaternions in order to avoid singularities in the object angular velocity mapping. Starting from that the controller has been extended employing an adaptive formulation to estimate the dynamical parameters of arms and object. Finally a model-free approach has been adapted form [19]. All controllers use load sharing coefficients to take in to account of possible differences in the arms strength.

Suitable software have been developed in order to validate the proposed approaches through simulations and experiments, confirming the results obtained in the theory. More precisely we have been able to *i*) confirm the effectiveness of the load sharing coefficients, *ii*) confirm the problems generated by a non perfect parametrization of the dynamical model, and hence *iii*) to develop an adaptive formulation in order to estimate in real time the model's dynamical parameters and *iv*) to provide a model-free approach with desired transient and steady state performance for the cooperative manipulation of an object.

Regarding the quaternion based approach, future efforts will be devoted toward incorporating kinematic uncertainties associated with the location of the object's center of mass, external disturbances, non rigid grasps as well as singularity avoidance.

Concerning the prescribed performance implementation future efforts will be devoted towards addressing non-rigid grasping and kinematic singularity avoidance.

Finally future efforts in the code developing could extend the driver to be used for joint position and velocity control in order to fully exploit the arm potential, moreover a filtering function can be added to mitigate reading noises.

Bibliography

- [1] S. A. Schneider and R. H. Cannon. Object impedance control for cooperative manipulation: Theory and experimental results. *IEEE, Transactions on Robotics and Automation*, 1992.
- [2] Y.-H. Liu, S. Arimoto, and T. Ogasawara. Decentralized cooperation control: non-communication object handling. *ICRA*, 1996.
- [3] Y.-H. Liu and S. Arimoto. Decentralized adaptive and nonadaptive position/force controllers for redundant manipulators in cooperations. *IJRR*, vol. 17, no. 3, pp. 232–247, 1998.
- [4] M. Zribi and S. Ahmad. Adaptive control for multiple cooperative robot arms. *CDC*, pp. 1392–1398, 1992.
- [5] O. Khatib, K. Yokoi, K. Chang, D. Ruspini, R. Holmberg, A. Casal. Decentralized cooperation between multiple manipulators. *International Workshop on Robot and Human Communication*, 1996.
- [6] F. Caccavale, P. Chiacchio, and S. Chiaverini. Task-space regulation of cooperative manipulators. *Automatica*, 1996.
- [7] J. Gudino-Lau, M. A. Arteaga, L. A. Munoz, and V. Parra-Vega. On the control of cooperative robots without velocity measurements. *TCST*, vol. 12, no. 4, pp. 600–608, 2004.
- [8] F. Caccavale, P. Chiacchio, A. Marino, and L. Villani. Six-dof impedance control of dual-arm cooperative manipulators. *Transactions On Mechatronics*, 2008.
- [9] D. Heck, D. Kostić, A. Denasi, and H. Nijmeijer. Internal and external force-based impedance control for cooperative manipulation. *ECC*, 2013.
- [10] S. Erhart and S. Hirche. Adaptive force/velocity control for multirobot cooperative manipulation under uncertain kinematic parameters. *IROS*, 2013.
- [11] J. Szewczyk, F. Plumet, and P. Bidaud. Planning and controlling cooperating robots through distributed impedance. *JRS*, 2002.
- [12] A. Tsiamis, C. K. Verginis, C. P. Bechlioulis, and K. J. Kyriakopoulos. Cooperative manipulation exploiting only implicit communication. *IROS*, 2015.
- [13] F. Ficuciello, A. Romano, L. Villani, and B. Siciliano. Cartesian impedance control of redundant manipulators for human-robot comanipulation. *IROS*, 2014.

Bibliography

- [14] A. Ponce-Hinestrosa, J. Castro-Castro, H. Guerrero-Reyes, V. Parra-Vega, and E. Olguyen-Dyaz. Cooperative redundant omnidirectional mobile manipulators: Model-free decentralized integral sliding modes and passive velocity fields. *ICRA*, 2016.
- [15] W. Gueaieb, F. Karray, and S. Al-Sharhan. A robust hybrid intelligent position/force control scheme for cooperative manipulators. *Transactions on Mechatronics*, 2007.
- [16] M. Ciocarlie, F. M. Hicks, R. Holmberg, J. Hawke, M. Schlicht, J. Gee, S. Stanford, and R. Bahadur. The velo gripper: A versatile single-actuator design for enveloping, parallel and fingertip grasps. *The International Journal of Robotics Research*, 2014.
- [17] Caccavale, F., Chiacchio, P., and Chiaverini, Task-space regulation of cooperative manipulators. *Automata*, 2000.
- [18] L. V. G. O. Bruno Siciliano, Lorenzo Sciavicco. *Robotics: Modelling, Planning and Control*. 1439-2232. Springer, 2010.
- [19] G. A. R. Charalampos P. Bechlioulis. A low-complexity global approximation-free control scheme with prescribed performance for unknown pure feedback systems. *Automatica*, 50(4):1217–1226, 2014.
- [20] T. H. Robert G. Bonitz. Force decomposition in cooperating manipulators using the theory of metric spaces and generalized inverses. *IEEE*, 1994.
- [21] WidowX Mark II arm web page:
<http://www.trossenrobotics.com/WidowxRobotArmMK2>
- [22] Robotis' Dynamixel series web page:
http://en.robotis.com/index/product.php?cate_code=101010
- [23] Coppelia Robotics web page:
<http://www.coppeliarobotics.com/index.html>
- [24] Coppelia Robotics: Building a V-REP clean model tutorial:
<http://www.coppeliarobotics.com/helpFiles/en/buildingAModelTutorial.htm>

