Politecnico di Torino

Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica
A.A. 2020/2021
Luglio 2021

Asynchronous Embedded Model Control for Networked Control Systems

Differential Wheeled Robot Application

Supervisors

Prof. Carlo NOVARA

PhD Carlos N. PEREZ MONTENEGRO

Candidate

Emanuele MANCIURIA - s262643

Politecnico
di Torino

Abstract

The aim of this thesis is to exploit the *Embedded Model Control* (EMC) in order to deal with *Networked Control Systems* (NCS), whose peculiarity is the usage of a communication network (CN) to connect the plant to the control unit, leading to *'asynchronous'* measurements and commands. In these types of systems the execution of the control law, implemented remotely, is driven by the arrival of the measures, whose frequency is non-deterministic and variable over time, implying that the controller must deal with variable sampling times.
Thus the canonical EMC design needs to be extended in order to face the problems provided by introducing a CN in a closed-loop control system, such extension assumes the name of *Asynchronous-EMC* (AEMC). More specifically, in this thesis, the effectiveness of the proposed control methodology is tested by applying it to a remote control of a two-wheeled differential drive mobile robot, endowed with a Raspberry Pi board. These tests require the construction of a CN between the robot and a laptop (which will act as a server). For this purpose two types of communication will be tested: the first is based on a p2p layout, built through an Ethernet cable connection, while the other uses a router for forwarding packets from the robot to the server and vice versa. Both of them are tested in order to provide more than one benchmark for the AEMC.

Acknowledgements

ACKNOWLEDGMENTS

Innanzitutto vorrei ringraziare il professore Carlo Novara per avermi dato la possibilità di cimentarmi con questa tesi ed il PhD Carlos Perez che mi ha pazientemente seguito durante il lavoro svolto.
Inoltre i miei più grandi ringraziamenti vanno alla mia ragazza, Maria, alla mia famiglia e a tutti i miei amici, i quali mi hanno dato sostegno morale e psicologico tale da permettermi di superare questi sei splendi seppur duri anni di università con la giusta motivazione.

First of all I would like to thank Professor Carlo Novara for giving me the opportunity to try my hand at the work of this thesis and the PhD Carlos Perez who patiently followed me throughout this work.
Moreover my biggest thanks go to my girlfriend, Maria, my family and all my friends, who gave me moral and psychological support which allowed me to overcome these six splendid but tough years of university with the right motivation.

Table of Contents

List of Tables VI

List of Figures VII

1 Introduction 1
- 1.1 Networked Control Systems Overview . . . 1
- 1.2 Literature Background . . . 3
- 1.3 Thesis Workflow . . . 4

2 Embedded Model Control Overview 5
- 2.1 Embedded Model Control Design Outlines . . . 5
  - 2.1.1 Embedded Model . . . 6
  - 2.1.2 Measurement Law . . . 8
  - 2.1.3 Control Law . . . 11
- 2.2 EMC Applied to a Mechanical Arm Driven by DC Motor . . . 12
  - 2.2.1 Case Study Simulations . . . 17
- 2.3 Asynchronous EMC . . . 20
  - 2.3.1 Asynchronous Embedded Model . . . 21
  - 2.3.2 Asynchronous Noise Estimator and Control Law . . . 22
- 2.4 Asynchronous EMC Applied to a Mechanical Arm Driven by DC Motor . . . 23
  - 2.4.1 Simulations Results . . . 24

3 Network Architectures 28
- 3.1 Packet Capture Libraries . . . 28
  - 3.1.1 Libpcap and WinPcap architectures . . . 29
  - 3.1.2 Libpcap functions . . . 30
  - 3.1.3 Performances Evaluation: p2p connection . . . 32
- 3.2 UDP Sockets . . . 34
  - 3.2.1 Unix native sockets libraries . . . 35
  - 3.2.2 QUdpSocket . . . 36

3.2.3 Performance Evaluation: UDP connection 38

4 GoPiGo3: Mobile Robot 41
4.1 Hardware Specifications 41
4.1.1 Actuators Study 44
4.1.2 Sensors Study 45
4.2 Software Specifications 48
4.2.1 GoPiGo3 `C++` libraries 48

5 Model and Control of Motor-Wheel System 52
5.1 Motor-Wheel Model Definition 52
5.1.1 Model Evaluation 54
5.2 Asynchronous-EMC Design for Motor-Wheel System 57
5.2.1 Embedded Model Design 58
5.2.2 Reference Generator Design 60
5.2.3 Noise Estimator Design 61
5.2.4 Control Law Design 62

6 Wheels Rotational Speed Control Simulations 63
6.1 Motor-Wheel Extended Plant Simulator 64
6.2 EMC for Simulated Motor-Wheel System:
Constant Sampling Time 67
6.2.1 Simulation 1 68
6.2.2 Simulation 2 69
6.2.3 Simulation 3 69
6.2.4 Results summary 70
6.3 EMC for Simulated Motor-Wheel System:
Constant Sampling Time 71
6.3.1 Networked Control System simulation 71
6.3.2 Simulation 1: $\sigma = 5\,\text{ms}$ 73
6.3.3 Simulation 2: $\sigma = 10\,\text{ms}$ 73
6.3.4 Simulation 3: $\sigma = 15\,\text{ms}$ 74
6.3.5 Results summary 74

7 Wheels Rotational Speed Control Experimental Tests 77
7.1 EMC for Real Motor-Wheel System:
On Board Control 78
7.1.1 Wheels on Board Control Tests General Settings 78
7.1.2 Wheels on board control: test 1 80
7.1.3 Wheels on board control: test 2 82
7.1.4 Wheels on board control: test 3 84
7.1.5 Results summary 85

7.2 EMC for Real Motor-Wheel System: Remote Control . . . 86
7.2.1 Wheels Remote Control Tests General Settings . . . 86
7.2.2 Wheels remote control, p2p connection: test 1 . . . 87
7.2.3 Wheels remote control, p2p connection: test 2 . . . 90
7.2.4 Wheels remote control, WiFi connection: test 1 . . . 92
7.2.5 Wheels remote control, WiFi connection: test 2 . . . 94
7.2.6 Results Summary . . . 95
7.3 EMC for Real Motor-Wheel System: Remote Control with Packet Dropout . . . 96
7.3.1 Wheel Remote Control With Losses: test 1 . . . 97
7.3.2 Wheel Remote Control With Losses: test 2 . . . 98
7.3.3 Wheel Remote Control With Losses: test 3 . . . 99
7.3.4 Wheel Remote Control With Losses: test 4 . . . 100
7.3.5 Wheel Remote Control With Losses: test 5 . . . 101
7.3.6 Results Summary . . . 101
7.4 Comparison with PID Controller . . . 102
7.4.1 PID and EMC controllers comparison: test 1 . . . 103
7.4.2 PID and EMC controllers comparison: test 2 . . . 103
7.4.3 Results Summary . . . 103

8 Conclusions 105

Acronyms 108

Bibliography 110

List of Tables

2.1 Fine Model Parameters 17

5.1 DC Motor Parameters 55

6.1 EMC Observer and Controller eigenvalues for motor-wheel simulations 68
6.2 EMC Observer, Controller and Reference Generator eigenvalues . . 72

7.1 EMC Observer, Controller and Reference Generator eigenvalues for right wheel speed control tests 78
7.2 EMC Observer, Controller and Reference Generator eigenvalues for left wheel speed control tests 79
7.3 EMC Observer, Controller and Reference Generator eigenvalues for right wheel speed remote control tests 86
7.4 EMC Observer, Controller and Reference Generator eigenvalues for left wheel speed remote control tests 87

List of Figures

1.1 Networked Control System conceptual scheme 1
1.2 effect of the CN process on a packet streaming 2

2.1 EMC general scheme 6
2.2 mechanical part of the system 12
2.3 block scheme of the designed Reference Dynamics 15
2.4 block scheme of the designed Noise Estimator 16
2.5 simulation 1, $\underline{e}$ 18
2.6 simulation 1, $\bar{e}$ 18
2.7 simulation 2, $\underline{e}$ 19
2.8 simulation 2, $\bar{e}$ 19
2.9 simulation 3, $\underline{e}$ 19
2.10 simulation 3, $\bar{e}$ 19
2.11 sample arrival times with variable intervals 20
2.12 Variability of the Sampling Times 25
2.13 model and plant outputs tracking the reference (green) following the trajectory of $\underline{y}$ (red) 26
2.14 model (top) and tracking (bottom) errors 26
2.15 reference generator output command (blue line), tracking error component (red line) and disturbance rejection component (yellow line) 27
2.16 state observer discrete time poles evolution 27

3.1 libpcap architecture 29
3.2 WinPcap architecture 29
3.3 packet format 33
3.4 test result: Round Trip Time values 34
3.5 signals and slots conceptual architecture 37
3.6 RTT values for UDP communication 39

4.1 back view of a GoPiGo3 3D model 41

4.2 front view of a GoPiGo3 3D model . . . 41
4.3 GoPiGo3 board: bottom side . . . 42
4.4 GoPiGo3 board: top side . . . 42
4.5 GoPiGo3 H-bridge integrated circuit scheme . . . 43
4.6 wheels angular speed (red) according to motor input voltage (blue) 44
4.7 wheels direct (red) and derived (blue) angular speed measurements with $T_s = 10\,ms$. . . . 46
4.8 wheels direct (red) and derived (blue) angular speed measurements with $T_s = 20\,ms$. . . . 47
4.9 wheels direct (red) and filtered derived (blue) angular speed measurements with $T_s = 10\,ms$. . . . 47

5.1 scheme of DC motor and wheel system . . . 53
5.2 block scheme of DC motor and wheel system . . . 54
5.3 input voltage obtained by ranging PWM duty cycle from 30% to 75% with increment of 5% each jump. . . . 55
5.4 comparison between measured wheels angular speed output (blue) and model output (red) for stairs input. . . . 56
5.5 input voltage obtained by PWM duty cycle values sequence: {0, 50, -50, 0}%. . . . 57
5.6 comparison between measured wheels angular speed output (blue) and model output (red) for double port input. . . . 57
5.7 Reference Generator with saturation of input $\underline{u}$ and state-feedback control. . . . 60

6.1 Simulink wheel-motor dynamics block scheme implementation . . 64
6.2 actuator implementation in Simulink . . . 65
6.3 input (red) and output (blue) of the simulated actuator. . . . 65
6.4 Simulink model for derived measurement method emulation . . . 66
6.5 Extended Plant voltage input (left) and angular speed outputs (right) 67
6.6 simulation 1 results . . . 68
6.7 simulation 2 results . . . 69
6.8 simulation 3 results . . . 70
6.9 Simulink block scheme of square wave generator for Triggered Subsystem block. . . . 72
6.10 simulation 1: extended plant outputs (top), model errors (bottom left) and tracking errors (bottom right) in 30 different realizations of T_i, with $\sigma = 5\,ms$. . . . 73
6.11 simulation 2: extended plant outputs (top), model errors (bottom left) and tracking errors (bottom right) in 30 different realizations of T_i, with $\sigma = 10\,ms$. . . . 74

6.12 simulation 3: extended plant outputs (top), model errors (bottom left) and tracking errors (bottom right) in 30 different realizations of T_i, with $\sigma = 15\,ms$. 75

7.1 left wheel, step response with eigenvalues in Tab. 7.1 79
7.2 right wheel, step response with eigenvalues in Tab. 7.1 79
7.3 right motor test 1 results 80
7.4 left motor test 1 results 81
7.5 right motor test 2 results 82
7.6 left motor test 2 results 83
7.7 right motor test 3 results 84
7.8 left motor test 3 results 85
7.9 right wheel remote control, p2p connection, test 1 results 88
7.10 left wheel remote control, p2p connection, test 1 results 89
7.11 right wheel remote control, p2p connection, test 2 results 90
7.12 left wheel remote control, p2p connection, test 2 results 91
7.13 right wheel remote control, WiFi connection, test 1 results 92
7.14 left wheel remote control, WiFi connection, test 1 results 93
7.15 right wheel remote control, WiFi connection, test 2 results 94
7.16 left wheel remote control, WiFi connection, test 2 results 95
7.17 remote control on WiFi connection, with 2% packet lost 97
7.18 remote control on WiFi connection, with 5% packet lost 98
7.19 remote control on WiFi connection, with 10% packet lost 99
7.20 remote control on WiFi connection, with 20% packet lost 100
7.21 remote control on WiFi connection, with 33.3% packet lost 101
7.22 PID (left) and EMC (right) remote control comparison, test 1 . . . 103
7.23 PID (left) and EMC (right) remote control comparison, test 2 . . . 104

Chapter 1

Introduction

1.1 Networked Control Systems Overview

In the last decade, *Networked Control Systems* (NCSs) have become increasingly popular in the technological field. An NCS can be defined as a control system whose feedback is closed via communication network (CN) [1], which is a communication channel shared among nodes which are potentially unrelated to the control system (see Fig. 1.1). The growing importance of these systems must be

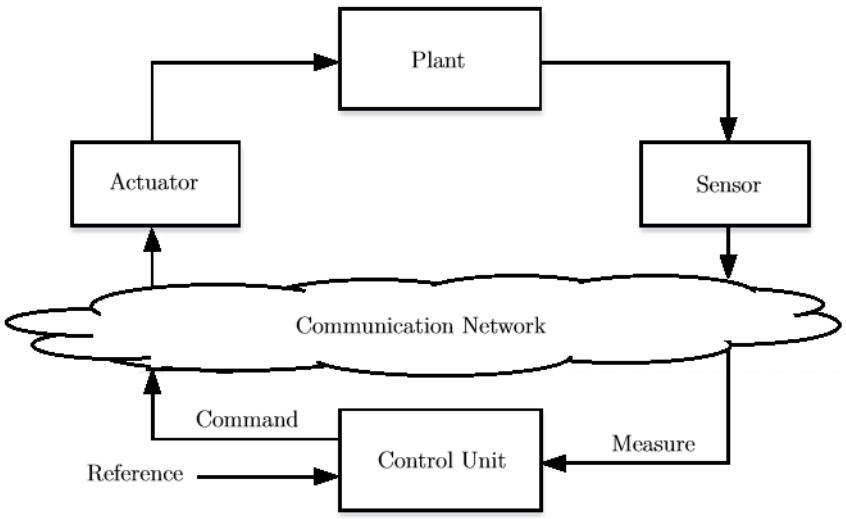


Figure 1.1: Networked Control System conceptual scheme

attributed to a considerable saving in infrastructure costs, allowed by the usage of wireless connections. Furthermore, there are also huge advantages coming from an easier sharing of information between devices belonging to the same controlled

environment, letting the data from different devices to be merged and, then, to be exploited in order to take smarter decisions over a larger physical space. In addition, the possibility of moving heavy calculations to remote servers also allows to have devices with less computing power and therefore less expensive. The applications of the NCS are very large and range from space and terrestrial exploration for hazardous environments to the automation of factories, but they also can be used on domestic robots or for remote diagnostics of automobiles opening up to the possibility of use in the Internet of Things (IoT), starting to talk about cloud control.
Nevertheless, it is undeniable that the NCS introduces some critical issues with respect to a traditional control system. Leaving aside all the problems related to reliability and security of communication, which are still fundamental aspects but which are not accounted in this thesis, the main problem is in the timing. In fact, regardless of the frequency with which the sensor sends measurements to the control unit, they arrive at totally variable time intervals, due to the NC that inserts not only transmission delays but also variations of the transmission delays (see Fig. 1.2). Therefore this type of systems are said to be with *asynchronous* measurements and commands. In this scenario it becomes very interesting to

Figure 1.2: effect of the CN process on a packet streaming

analyze the study, whose source is reported here [2], of a possible application to this type of systems of a control design methodology called **Embedded Model Control** (EMC). The choice of the EMC is justified by the fact that this control method results to be very precise since it is based on the rejection of disturbances, allowing to zero (or strongly reduce) errors coming from inaccurate parameters, unmodeled dynamics or generic causal disturbances acting on the plant. In addition, as it will seen in following sections, this control methodology results to be very light in computational terms. Thus, by considering time delays or data loss as a source of disturbance, it is possible to think about EMC as a method able to reject these disturbances too. Anyway, the traditional design of the EMC found in the literature is not sufficient to deal with the asynchronous measurements and commands, thus, an extension of this method is needed. To this aim it is introduced the **asynchronous-EMC** (AEMC), which extends the canonical EMC such as to make it time-adaptive.

The goal of this thesis is to validate the theoretical results related to AEMC by applying it to a remote control of a two-wheeled differential drive mobile robot.

1.2 Literature Background

In recent years, research on NCS has led to the study of a large variety of control methodologies to be applied on it. One of the most used approaches is the optimal control, in particular the MPC, which is a model-based optimal control whose purpose is to predict the evolution of the model for a desired number of steps, in which the "most suitable" command to be supplied to the plant is computed. The most suitable command is the result of an optimization problem, for a certain cost function which includes indices such the tracking error and the command action. An example of the applications of the MPC on the NCSs is provided in [3], where some precautions are used to deal with CN issues. Anyway the MPC is strongly affected to parameter errors, moreover this kind of control is very heavy in computational terms, therefore it may not be suitable for applications in which a very low sampling time is required.
Other solutions involving optimal control are purposely studied for the NCSs communication characteristics. One example is given in [4] where the methodology named packet-based control is employed. The idea behind this control methodology is to provide a set of control predictions and the to select the most suitable according to the network conditions. Another example of control specific for NCSs is described in [5] where the Decentralized-MPC is explored as possible solution. It concerns the usage of a hierarchical structure between the controllers, one central MPC on a remote server and several distributed MPCs on the controllable system. Anyway all these solutions are very complex and requires a quite high computational effort even on the controlled nodes.
In any case, model-based controls are not the only control methodologies to be used in these applications, in fact a wide part of the research on NCS focuses on the use of PID controllers, highlighting their effectiveness despite the very low design complexity. The possibility of the application of a PID controller on the NCSs is explored in [6], where it is remarked that a good design and tuning procedure is sufficient to have acceptable results in the control of remote plant. Anyway the PID controller suffers some problems of robustness which makes it not very suitable for an application where the system can be subject to many disturbances coming from external factors. A comparison between two remote controls performed with an EMC controller and with a PID one is provided at the end of the thesis.

1.3 Thesis Workflow

As previously mentioned, the focus of the thesis is on the construction of a remote control system, based on the EMC design methodology, to be applied to a mobile robot. In order to achieve this goal, first of all there will be a general overview of the theory regarding EMC. In fact, in chapter 2, the key points for its design are deepened and a simple practical application of a case study is performed through a simulation. After that the transition to asynchronous-EMC is faced, dealing with the main changes to be made to the traditional design so that it can manage asynchronous measures and commands. Also in this case, another simulation is performed in order to practice these AEMC design notions, thus concluding the theoretical part.
In chapter 3 we talk about how to build communication on the network so that the robot and my own laptop (n.b. it is often referred to as "server") can exchange data one each other. To this aim two kinds of network architectures are described, highlighting the tools employed to implement them and, once set up, evaluating both by means of practical tests.
Then the chapter 4 introduces the robot used for this project, illustrating both its hardware and software features. In addition, the programming tools that the vendors provide to developers are also shown, emphasizing those used within the project.
Finally, the time comes to apply the theoretical results on mobile robot physical system. In order to do this it is needed to start from the wheels control since, once it is achieved, it is possible at least to obtain a very raw control of the robot movements simply by providing suitable speed references to the wheels. Thus in chapter 5 the model of the motor-wheel system is provided, in addition a deep study of actuators and sensors is shown. This is needed both for the asynchronous-EMC design and for the simulation of the extended plant. After that, in chapter 6, the AEMC designed for the motor-wheel system is tested. First it is used for simulations, performed by means of Simulink framework, evaluating the control design for both constant and variable sampling times; then the AEMC design is "translated" in `C++` code, allowing to test it on the physical system. The tests include both the on board and the remote control of the wheels, and for the remote control both the network architectures are tested.

Chapter 2

Embedded Model Control Overview

The principles of the EMC are based on the computation of the expected response of the system to a certain command $u(t)$ by means of a run-time simulation of the model, whose result must be compared to the measured output of the plant obtained by applying the same command $u(t)$. This comparison determines the *model error*, which is used to estimate non-causal and unpredictable signals acting on the system called ***driving noises***; the number of driving noises and their effects on the states of the system is matter of the design procedure.
Nevertheless the noise must be distinguished from all the possible causal disturbances acting on the real plant, which, due to their causality, can be added to the model as *states*, which can't be controlled by the command u. This provides a state-space $\mathbf{x}(t)$ composed by $\mathbf{x}_c(t)$ and $\mathbf{x}_d(t)$ for which, at design time, can be defined the dependence on the driving noise.
The computation of the disturbances is the core of the EMC design method since they are used to adjust the controller *command* so that the disturbance rejection can be achieved. The end result is a control methodology very precise, with imperfect models too, without renounce to the efficiency.
It will follow a more detailed description of EMC's design method and a study of its use in systems with asynchronous measurements and commands, after that an analysis of a case-study which allows to better understand the effectiveness of this methodology on those kind of systems.

2.1 Embedded Model Control Design Outlines

For the description of the *EMC* design methodology is used the source [7] from which are taken the diagram in figure below and which will be summed up with

the following sections.

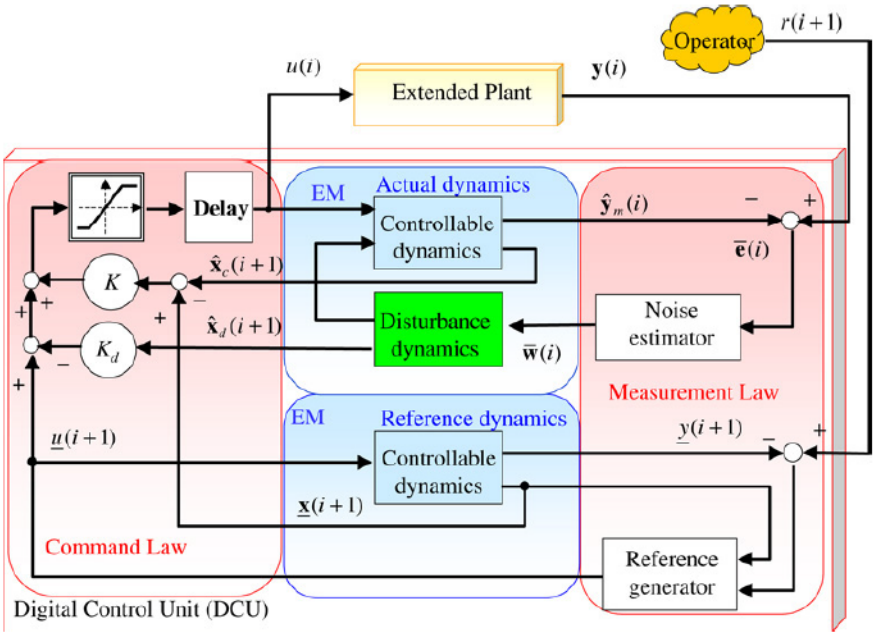


Figure 2.1: EMC general scheme

It is possible to recognise in the figure 2.1 the two main blocks which are the fundamentals of the control theory, which are:

- **Extended Plant**, that is the real *continuous* system to control, therefore it contains all the physical processes, neglected dynamics and unpredictable noises included. It is the block between the signals u and y.
- **Control Unit** (CU), implemented in a digital environment which includes all the modules used for the design of the EMC.

The CU can be further divided in other several units that are the **Embedded Model** (EM), the **Measurement Law** and the **Control Law**.

2.1.1 Embedded Model

It is possible to recognise two different parts belonging to the EM that are the *Actual dynamics* and the *Reference dynamics*, the first one must be the Extended Plant copy and its goal is to replicate as best as possible the plant behavior, while

the second one is introduced to track the reference $\mathbf{r}$ which could include requests that cannot be satisfied by the system and which will be referred to as *'operator requests'*.
In the *Actual dynamics* the state-space $\mathbf{x}$ is considered to be composed by both the controllable state $\mathbf{x}_c$ and the disturbance state $\mathbf{x}_d$, which is not reachable by the command input $\mathbf{u}$, nevertheless it is a way to model the *causal* disturbances acting on the model such as *neglected dynamics* or *parametric uncertainties*, then the action of the disturbance $\mathbf{w}$ is modelled so that the state $\mathbf{x}$ linearly depends on it. Finally the state-space equations, in *continuous time* (CT), are[1]

$$\dot{\mathbf{x}}(t) = A_{CT}\mathbf{x}(t) + B_{CT}\mathbf{u}(t) + G_{CT}\mathbf{w}(t), \qquad \mathbf{x}(0) = \mathbf{x}_0. \tag{2.1}$$

Given the *sample time* T, a possible discretization of this model can be done by approximating the derivative with the *forward Euler method*

$$\dot{\mathbf{x}}(t_k) \simeq \frac{\mathbf{x}(t_k + T) - \mathbf{x}(t_k)}{T}, \quad t_k = kT, \quad for \;\; k \in \mathbb{N} \tag{2.2}$$

and by defining the following matrices

$$A = TA_{CT} + I_{n_x \times n_x}, \quad B = TB_{CT}, \quad G = TG_{CT}. \tag{2.3}$$

it is possible to obtain the equations in discrete time[2]

$$\begin{aligned} &\begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_d \end{bmatrix}(k+1) = \mathbf{x}(k+1) = A\mathbf{x}(k) + B\mathbf{u}(k) + G\mathbf{w}(k), \qquad \mathbf{x}(0) = \mathbf{x}_0, \\ &\mathbf{z}_m(k) = F\mathbf{x}(k) \\ &\mathbf{y}_m(k) = C\mathbf{x}(k). \end{aligned} \tag{2.4}$$

Since $\mathbf{x}$ can be splitted in $\mathbf{x}_c$ and $\mathbf{x}_d$, dimensioned respectively n_c and n_d, it is possible to consider the following block division in the matrices:

$$\begin{aligned} &A = \begin{bmatrix} A_c & H_c \\ 0_{n_d \times n_c} & A_d \end{bmatrix}, \quad B = \begin{bmatrix} B_c \\ 0_{n_d \times n_u} \end{bmatrix}, \quad G = \begin{bmatrix} G_c \\ G_d \end{bmatrix}, \\ &C = \begin{bmatrix} C_c & C_d \end{bmatrix}, \quad F = \begin{bmatrix} F_c & 0_{n_z \times n_d} \end{bmatrix}. \end{aligned} \tag{2.5}$$

The matrices must be constructed such that both the pairs (A_c, B_c) and (A, G) are controllable, and both the pairs (F_c, A_c) and (C, A) are observable. In simpler cases it is assumed $\mathbf{z}_m = \mathbf{y}_m$ hence $C = F$.
Now it is possible to distinguish two different dynamics:

---

[1]For simplicity the considered plant is an LTI system, however the results can be extended to the more general case of non-linear systems exploiting linearization methods

[2]for the sake of brevity the time $t_k = kT$ is replaced by the more compact notation k, assuming the meaning of "step", thus the following steps are denoted as $k + i$, $i \in \mathbb{N}^+$

- ***Disturbance Dynamics***, which takes as input the driving noise $\mathbf{w}$ and provides both the disturbance state $\mathbf{x}_d$ and the disturbance $\mathbf{d}$, which cumulates all the disturbances acting on the controllable state;

$$\begin{aligned} &\mathbf{x}_d(k+1) = A_d\mathbf{x}_d(k) + G_d\mathbf{w}(k), \qquad \mathbf{x}_d(0) = 0 \\ &\mathbf{d}(k) = H_c\mathbf{x}_d(k) + G_c\mathbf{w}(k) \end{aligned} \tag{2.6}$$

- ***Controllable Dynamics***, here the inputs are the command $\mathbf{u}$ and the disturbance d and are computed both the evolution of the controllable state and the model output. In this case $C = F$, hence $C_d = 0$ and $F_c = C_c$:

$$\begin{aligned} &\mathbf{x}_c(k+1) = A_c\mathbf{x}_c(k) + B_c\mathbf{u}(k) + \mathbf{d}(k), \qquad \mathbf{x}_c(0) = \mathbf{x}_{c0} \\ &\mathbf{y}_m(k) = C_c\mathbf{x}_c(k) \ . \end{aligned} \tag{2.7}$$

The second part of the EM is the *Reference dynamics*, where are proposed the same equations of Controllable dynamics, with the difference that, in this case, the disturbance isn't considered:

$$\begin{aligned} &\underline{\mathbf{x}}(k+1) = A_c\,\underline{\mathbf{x}}(k) + B_c\,\underline{\mathbf{u}}(k), \quad \underline{\mathbf{x}}(0) = \underline{\mathbf{x}}_0 \\ &\underline{\mathbf{y}}(k) = C_c\,\underline{\mathbf{x}}(k) \ . \end{aligned} \tag{2.8}$$

The input $\underline{\mathbf{u}}$ is the output of the *Reference Generator*, and it represents the input to provide to the disturbance-free model such that the output $\underline{\mathbf{y}}$ is able to track the *operator request* ($\mathbf{r}$), complying with the eventual constraints imposed on both command and states. For this reason, $\underline{\mathbf{u}}$ is typically the result of a further control law applied to the system represented in 2.8, in this way the designer is also able to set the desired behavior of the reference.

2.1.2 Measurement Law

As told in section 2.1 the noise $\mathbf{w}$ is unpredictable and command independent, thus it is not possible to do any assumption on the values $\mathbf{w}(k+i)$, with $i > 0$; it is only allowed to *estimate* its value at the actual time step k in function of the *model error* $\mathbf{e}(k) = \mathbf{y}(k) - \mathbf{y}_m(k)$, where $\mathbf{y}$ is the measurement of the *Extended Plant*.
The ***Noise estimator*** block (in figure 2.1) is in charge to do this work and its output, which is the estimate of the noise, is named $\bar{\mathbf{w}}$. This signal is used as input of the *Disturbance dynamics*, in this way through *Actual dynamics* and *Noise estimator* a loop is constructed, which implements, in practice, a ***state predictor***, since it provides the one-step prediction $\hat{\mathbf{x}}(k+1)$. The model error will be denoted as $\bar{\mathbf{e}}(k) = \mathbf{y}(k) - \hat{\mathbf{y}}_m(k)$, where $\hat{\mathbf{y}}_m(k)$ is the estimate of the plant output.
The *estimator* of $\mathbf{w}$ is written, in z domain as

$$\bar{\mathbf{w}}(z) = \mathbf{L}(z)\bar{\mathbf{e}}(z) \tag{2.9}$$

but the matrix $\mathbf{L}(z)$ cannot be uniquely defined, unless some assumptions are made. In this section will be afforded the solution proposed in [7], where is exposed the general case of $n_y \geq 1$ and $n_w \geq 1$, with n_y and n_w respectively the dimensions of $\bar{e}$ and $\bar{w}$, under several assumptions that lead to a unique solution for the estimator. Let the matrix A to be an block upper-triangular matrix, if not it is possible to force it to satisfy this assumption. From A can be obtained m diagonal blocks, namely A_j, with $dim(A_j) = n_j \times n_j$, for $j = 0, ..., m-1 \leq n_y - 1$, and the consequent division of the state $\mathbf{x}$ in m sub-states named $\mathbf{x}_j$, each one of dimension n_j. In matrix C, by partitioning each row C_j in n_y blocks $C_{j,i}$, each one with a number of columns equal to the number of columns of the respective block A_i, can be extracted the matrices C_j corresponding to the diagonal blocks $C_{j,j}$; let's denote the measures corresponding to each row of C as y_j.
Result: if each couple (A_j, C_j) is observable, then the measures $y_0, ..., y_{m-1}$ are sufficient for the estimation of $\mathbf{w}$. The value of m must be as small as possible, the partitions must be unique, and it is required that $n_y = m$.
Assume, then to divide $\mathbf{w}$ in m sub-vectors $\mathbf{w}_j$, each sized n_{wj}, and to rearrange them so that, from the matrix G, are obtained diagonal blocks G_j, of dimension $n_j \times n_{wj}$, for which the couples (G_j, A_j) are controllable.
Under such assumptions estimator can be uniquely chosen, by placing the desired eigenvalues λ_m over the closed-loop system formed among *Actual dynamics* and *Noise estimator*. For the definition of the matrix $\mathbf{L}(z)$ the above definition of $\mathbf{w}_j$ must be employed, since it is useful to consider more than one channel of $\mathbf{L}$, each one dimensioned $n_{wj} \times 1$ and driven by the *scalar* value e_j like: $\bar{\mathbf{w}}_j = \mathbf{L}_j(z)\bar{e}_j$. As can be seen the general problem is reduced to several simpler ones and each of them falls in the case of $n_y = m = 1$.
Anyway, for the sake of readability, in the following considerations the "j" subscript will be neglected, and the problem will be recasted to the simpler case of a scalar output (i.e. $n_y = 1$). Note that, since this is also the case of each single channel, it is very simple to came back to the case $n_y > 1$, in fact it suffices to reiterate the following assumptions to each channel j and then, reconstruct the vector $\bar{\mathbf{w}}$ from the $\bar{\mathbf{w}}_j$.
By comparing n and n_w, respectively the dimensions of the state $\mathbf{x}$ and the noise $\mathbf{w}$, two different solutions for the estimator $\mathbf{L}$ may occur:

- if $n \leq n_w$ it is possible to use a ***static estimator*** (i.e. $\mathbf{L}(z) = L$) of the kind $\bar{\mathbf{w}}(k) = L\,\bar{e}(k)$; moreover this case is always recastable to $n = n_w$ by combining some components of $\mathbf{w}$. Thus the equations of the overall predictor are:

$$\begin{aligned} &\hat{\mathbf{x}}(k+1) = A\,\hat{\mathbf{x}}(k) + B\,\mathbf{u}(k) + G\,\bar{\mathbf{w}}(k), \quad \hat{\mathbf{x}}(0) = \hat{\mathbf{x}}_0, \\ &\bar{e}(k) = y(k) - \hat{y}_m(k), \\ &\bar{\mathbf{w}}(k) = L\,\bar{e}(k) \ . \end{aligned} \tag{2.10}$$

Applying the right substitutions it can be obtained

$$\hat{\mathbf{x}}(k+1) = (A - GLC)\,\hat{\mathbf{x}}(k) + B\,\mathbf{u}(k) + G\,L\,y(k), \quad \hat{\mathbf{x}}(0) = \hat{\mathbf{x}}_0, \tag{2.11}$$

where $A_m = A - GLC$. The matrix L must be designed such that the necessary condition $|\lambda_i(A_m)| < 1,\ \forall\ i = 1, ..., n$ is satisfied.

- if $n > n_w$ must be employed a ***dynamic estimator*** since the degrees of freedom offered by a static matrix L (of dimensions $n_w \times 1$) are not sufficient to place n poles, thus it is needed to add other parameters in order to accomplish the estimator design: the idea is to add as many *states* as the difference between the dimensions of x and w, and to define the estimate $\bar{\mathbf{w}}$ as a linear combination of $\bar{e}$ and the added states.
  Typically $n - n_w = 1$ since it gives the possibility to implement a filter for the undesired high-frequency disturbances. In this specific case the state is a scalar q dependent on, both, its past value and the model error $\bar{e}$. This yields to the equation

$$q(k+1) = A_e\,q(k) + L_e\,\bar{e}(k) \tag{2.12}$$

and the estimate can be:

$$\bar{\mathbf{w}}(k) = M_m\,q(k) + L\,\bar{e}(k)\ . \tag{2.13}$$

In z-domain, assuming $A_e = 1 - \beta$ and $L_e = 1$, the transfer function between $\bar{e}$ and q has the form of a filter, since results to be $q(z) = (z - 1 + \beta)^{-1}\,\bar{e}(z)$, hence the estimator has the following dynamic form:

$$\bar{\mathbf{w}}(z) = \mathbf{L}(z)\,\bar{e}(z) = \left(L + M_m \frac{1}{z - 1 + \beta}\right)\,\bar{e}(z)\ . \tag{2.14}$$

Now it is possible to define the overall closed loop state equations, which form the state predictor, as

$$\begin{aligned}
&\hat{\mathbf{x}}(k+1) = A\,\hat{\mathbf{x}}(k) + B\,\mathbf{u}(k) + G\,\bar{\mathbf{w}}(k), \quad \hat{\mathbf{x}}(0) = \hat{\mathbf{x}}_0,\\
&q(k+1) = (1-\beta)q(k) + \bar{e}(k),\\
&\bar{e}(k) = y(k) - \hat{y}(k)\\
&\bar{\mathbf{w}}(k) = M_m\,q(k) + L\,\bar{e}(k)\ .
\end{aligned} \tag{2.15}$$

Finally, doing some arrangements and substitutions, the state equations can be rewritten in matricial form using the augmented state $\mathbf{x}_I = \begin{bmatrix}\hat{\mathbf{x}}\\ q\end{bmatrix}$ in the following way:

$$\mathbf{x}_I(k+1) = \begin{bmatrix} A - GLC & GM_m \\ -C & 1-\beta \end{bmatrix} \mathbf{x}_I(k) + \begin{bmatrix} B & GL \\ 0_{1\times n_u} & 1 \end{bmatrix} \begin{bmatrix}\mathbf{u}(k)\\ y(k)\end{bmatrix}, \tag{2.16}$$

where $A_m = \begin{bmatrix} A - GLC & GM_m \\ -C & 1-\beta \end{bmatrix}$ must satisfy the condition $|\lambda_i(A_m)| < 1$, $\forall\ i = 1, ..., n+1$, that is achievable by properly designing the set of values $\{\beta, l_0, ..., l_{n_w}, m_0, ..., m_{n_w}\}$ in function of the placed eigenvalues $\lambda_{m,i}$. Note that the number of parameters may be oversized with respect to the number of poles to place, thus there can be parameters which must be set arbitrarily.

2.1.3 Control Law

Last but not least is the *Control Law*, which is constructed by correcting the reference $\underline{\mathbf{u}}$, provided by *Reference Generator*, with two terms depending by both the disturbance state $\mathbf{x}_d$ and the tracking error $\underline{\mathbf{e}}_c$, which is defined as

$$\underline{\mathbf{e}}_c(k) = \underline{\mathbf{x}}(k) - (\mathbf{x}_c(k) + Q\mathbf{x}_d(k)) \tag{2.17}$$

where $\underline{\mathbf{x}}$ is the reference state (from 2.8) and Q is a weight matrix. Hence can be written the evolution of $\underline{\mathbf{e}}_c$ by means of the equation

$$\begin{aligned} \underline{\mathbf{e}}_c(k+1) =& A_c\underline{\mathbf{e}}_c(k) + B_c(\underline{\mathbf{u}}(k) - \mathbf{u}(k)) - \\ & - (G_c + QG_d)\mathbf{w}(k) + \\ & + (A_cQ - H_c - QA_d)\mathbf{x}_d(k)\ . \end{aligned} \tag{2.18}$$

Since $\mathbf{x}_d$ and $\mathbf{w}$ can't be controlled and since both of them appear in 2.18, one solution to guarantee stability in the evolution of $\underline{\mathbf{e}}_c$ is to set to zero the related term. Finally the *control law* can be a linear function of the states like this

$$\mathbf{u}(k) = \underline{\mathbf{u}}(k) + K\underline{\mathbf{e}}_c(k) - M\mathbf{x}_d(k) \tag{2.19}$$

where can be distinguished the two terms mentioned before. The three matrices K, M and Q must be designed in order to satisfy the two necessary conditions:

$$1.\ \ |\lambda_k(A_c - B_cK)| < 1, \forall\ k = 1, ..., n_c\,, \tag{2.20}$$

$$2.\quad \begin{bmatrix} H_c + Q\,A_d \\ 0 \end{bmatrix} = \begin{bmatrix} A_c & B_c \\ F_c & 0 \end{bmatrix} \begin{bmatrix} Q \\ M \end{bmatrix}\ . \tag{2.21}$$

Thus, in practice, it is possible to compute K in function of the placed eigenvalues λ_c of the matrix in point 1; and then compute the matrices Q and M by solving the equations in point 2, also known as Davison-Francis relationship.

2.2 EMC Applied to a Mechanical Arm Driven by DC Motor

Here a simple example of EMC application will be explored with the aim to apply the theoretical results in the previous sections.Next step will be to perform suitable changes to the design in order to obtain an *asynchronous-EMC* which can handle the case of variable sampling time.
The case study consists in a rigid arm of length l and mass m and inertia J_a, which rotates in a vertical plane, driven by a DC motor. The goal is to control in position the arm.

Extended Plant Definition

The fine model only includes the mechanical part of the system described above, thus the control command is a torque C_m impressed to the motor side and transmitted by a gearbox reduction ratio τ to the load; this torque must rotate the motor shaft with an inertia J_m and affected by a friction with coefficient β_m. The transmitted torque C_a, at load side, moves the arm that is affected by a friction with coefficient β_a and by the gravity acceleration $g = 9.81\ \frac{m}{s^2}$.
Finally all its components are considered to be stiff.

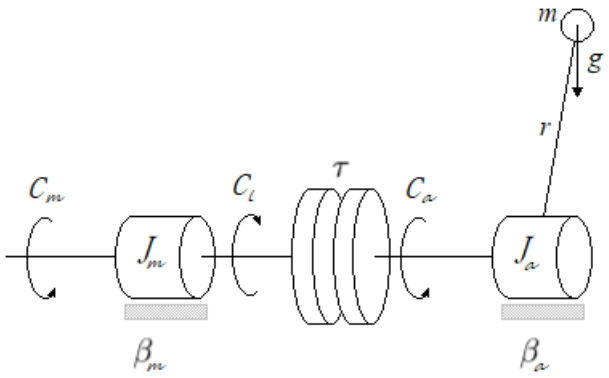


Figure 2.2: mechanical part of the system

The figure shows a scheme of the system just described, where the mass m represents the mass of the entire arm focused at the CoG, spaced r from the center of rotation. Defining the states θ and ω, respectively the angular position and the angular velocity of the arm, it is possible to set up the state equations starting from the

equilibrium of the torques, considering that at horizontal position $\theta = 0$.

$$\begin{aligned} &motor\ side: \\ &\qquad C_m = J_m\dot{\omega}_m(t) + \beta_m\omega_m(t) + C_l \\ &arm\ side: \\ &\qquad C_a = J_a\dot{\omega}(t) + \beta_a\omega(t) + r\,mg\,\sin\left(\frac{\pi}{2} - \theta(t)\right), \end{aligned} \tag{2.22}$$

since the relationships $C_l = C_a/\tau$ and $\omega_m = \omega_a\tau$ hold, by suitable substitutions and with some arrangements it is possible to write the following equation:

$$\frac{C_m}{\tau} = J_{eq}\dot{\omega}(t) + \beta_{eq}\omega(t) + \frac{r\,mg}{\tau^2}\sin\left(\frac{\pi}{2} - \theta(t)\right) \tag{2.23}$$

where

$$\begin{aligned} J_{eq} =& J_m + \frac{J_a}{\tau^2}, \quad J_a = \frac{1}{3}ml^2 \\ &\beta_{eq} = \beta_m + \frac{\beta_a}{\tau^2}. \end{aligned} \tag{2.24}$$

Finally, considering $x_1 = \theta$ and $x_2 = \omega$ and being C_m the command input u, the state space equations are

$$\begin{aligned} &\dot{x}_1(t) = x_2(t) \\ &\dot{x}_2(t) = -\frac{\beta_{eq}}{J_{eq}}x_2(t) - \frac{r\,mg}{\tau^2 J_{eq}}\sin\left(\frac{\pi}{2} - x_1(t)\right) + \frac{1}{\tau J_{eq}}u(t) \\ &\mathbf{x}(0) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}(0) = \mathbf{x}_0\ . \end{aligned} \tag{2.25}$$

Is worth to note that, although the system is nonlinear, with a sufficiently high reduction factor the nonlinear term is naturally damped since it is divided by τ^2. Anyway the goal of this simulation is to show the effectiveness of the noise estimation and the rejection of the disturbances, so the final results will highlight the difference between the EM implemented both with disturbance rejection and without it.

Actual Dynamics Design

In the EM the nonlinear term will be neglected, becoming an LTI system where both $J_{eq,m}$ and $\beta_{eq,m}$ differ from the respective real parameters J_{eq} and β_{eq} by introducing some parametric errors form of percentage factor.

Considering one single disturbance state and, by choosing $n_w = 2$ a *dynamical noise estimator* will be implemented. Finally, considering the angular position as both measurable output and performance output (i.e. $y_m = z_m = \theta$), and being $\mathbf{x} = \begin{bmatrix}\mathbf{x}_c \\ x_d\end{bmatrix}$, where $\mathbf{x}_c = \begin{bmatrix}\theta \\ \omega\end{bmatrix}$, the equations in CT are

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \begin{bmatrix}0 & 1 & 0 \\ 0 & -\dfrac{\beta_{eq,m}}{J_{eq,m}} & 1 \\ 0 & 0 & 0\end{bmatrix}\mathbf{x}(t) + \begin{bmatrix}0 \\ \dfrac{1}{\tau J_{eq,m}} \\ 0\end{bmatrix}u(t) + \begin{bmatrix}0 & 0 \\ 1 & 0 \\ 0 & 1\end{bmatrix}\mathbf{w}(t), \\ y_m(t) &= z_m(t) = \begin{bmatrix}1 & 0 & 0\end{bmatrix}\mathbf{x}(t), \quad \mathbf{x}(0) = \mathbf{x}_0\end{aligned} \tag{2.26}$$

and the discretization can be performed using the method in 2.3, which provides the following equations in DT:

$$\begin{aligned}\mathbf{x}(k+1) &= \begin{bmatrix}1 & T & 0 \\ 0 & -T\dfrac{\beta_{eq,m}}{J_{eq,m}} + 1 & T \\ 0 & 0 & 1\end{bmatrix}\mathbf{x}(k) + \begin{bmatrix}0 \\ T\dfrac{1}{\tau J_{eq,m}} \\ 0\end{bmatrix}u(k) + \begin{bmatrix}0 & 0 \\ T & 0 \\ 0 & T\end{bmatrix}\mathbf{w}(k), \\ y_m(k) &= z_m(k) = \begin{bmatrix}1 & 0 & 0\end{bmatrix}\mathbf{x}(k), \\ \mathbf{x}(0) &= \mathbf{x}_0\ .\end{aligned} \tag{2.27}$$

Hence the matrices are

$$\begin{gathered}A_c = \begin{bmatrix}1 & T \\ 0 & -T\dfrac{\beta_{eq,m}}{J_{eq,m}} + 1\end{bmatrix} \quad H_c = \begin{bmatrix}0 \\ T\end{bmatrix} \quad B_c = \begin{bmatrix}0 \\ T\dfrac{1}{\tau J_{eq,m}}\end{bmatrix} \quad G_c = \begin{bmatrix}0 & 0 \\ T & 0\end{bmatrix} \\ A_d = 1 \quad B_d = 0 \quad G_d = \begin{bmatrix}0 & T\end{bmatrix} \\ C_c = F_c = \begin{bmatrix}1 & 0\end{bmatrix} \quad C_d = F_d = 0\ ,\end{gathered} \tag{2.28}$$

once all the matrices are defined, the *Controllable* and *Disturbance dynamics* can be easily constructed as in 2.7 and 2.6 respectively.

Reference Generator Design

The *reference generator* is based on the equations described in 2.8 and its goal is to generate a reference $\underline{u}$ such that the output $\underline{y}$ follows the operator request r with a smooth dynamic, such that input and state constraints are respected. In this specific case study no input constraints are considered, while the state constraints are handled by controlling the dynamic of the system with a static state feedback control of the kind

$$\underline{u}(k) = K_{ref}\underline{\mathbf{x}}(k) + Nr(k) \tag{2.29}$$

where K_{ref} and N are chosen so that (i) $|\lambda_i(A_c - B_c K_{ref})| < 1, \forall\ i = 1, ..., n_c$, defining the components of K_{ref} in function of the placed eigenvalues $\lambda_{ref,i}$, and (ii) $N = \left[C_c[I - (A_c - B_c K_{ref})]^{-1} B_c\right]^{-1}$, which makes the output to track the reference r. By employing a state feedback control (figure 2.3), it is possible to set the desired dynamics to the reference, making it smoother than a step or an impulse given by the operator. In this example $n_c = 2$, thus it is needed to place as many poles, chosen arbitrarily such that it is respected the condition (i).
Let $\lambda_{ref,1}$ and $\lambda_{ref,2}$ be the placed poles, $K_{ref} = \begin{bmatrix} k_1 & k_2 \end{bmatrix}$ is computed as:

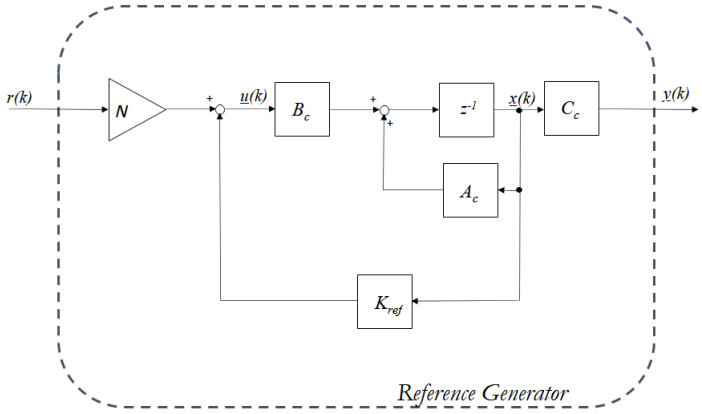


Figure 2.3: block scheme of the designed Reference Dynamics

$$k_1 = \frac{\lambda_{ref,1}\lambda_{ref,2} - a_{c,11}a_{c,22} + a_{c,11}b_{c,21}k_2}{a_{c,12}b_{c,21}}$$
$$k_2 = \frac{a_{c,11} + a_{c,22} - \lambda_{ref,1} - \lambda_{ref,2}}{b_{c,21}}$$

where $a_{c,ij}$ and $b_{c,ij}$ are respectively the components of A_c and B_c.

Noise Estimator Design

As written above, for this design, the *Noise estimator* is chosen to be *dynamic*, thus the predictor shown in the equation 2.16 is employed. In this case it needs to place four eigenvalues $\lambda_{m,i}$, in function of which it is possible to compute the parameters β, l_0, l_1, m_0, m_1 are found in function of the eigenvalues and the estimator can be finally constructed. Note that there is one free parameter which can be set to an arbitrary value, independently from the eigenvalues.
The figure 2.4 represents a possible block scheme for the designed Noise estimator,

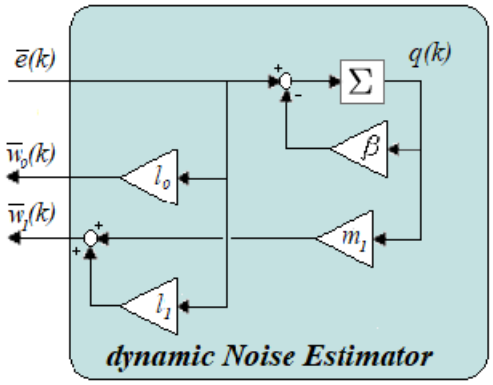


Figure 2.4: block scheme of the designed Noise Estimator

where the Σ block represents an integrator in DT. The figure also shows that for this design was set $m_0 = 0$, while other parameters are computed as follows:

$$\begin{aligned}
\beta &= a_{11} + a_{22} + a_{33} - \alpha_3 + 1 \\
l_0 &= \frac{1}{a_{12}g_{21}}((\beta - 1)(a_{11} + a_{22} + a_{33}) - a_{11}a_{22} - a_{11}a_{33} - a_{22}a_{33} + \alpha_2) \\
l_1 &= \frac{1}{a_{12}a_{23}g_{32}}(a_{11}a_{22}a_{33} - (\beta - 1)(a_{11}a_{22} + a_{11}a_{33} + a_{22}a_{33}) + \\
&\qquad + a_{12}g_{21}l_0(a_{33} - (\beta - 1)) - \alpha_1) \\
m_1 &= \frac{1}{a_{12}a_{23}g_{32}}((\beta - 1)(a_{11}a_{22}a_{33} + a_{12}a_{33}g_{21}l_0 - a_{12}a_{23}g_{32}l_1) + \alpha_0)
\end{aligned} \tag{2.30}$$

where α_0, α_1, α_2 and α_3 are the coefficient of the desired *characteristic polynomial*

$$P(\lambda) = \lambda^4 - \alpha_3\lambda^3 + \alpha_2\lambda^2 - \alpha_1\lambda + \alpha_0$$

and

$$\begin{aligned}
\alpha_0 &= \lambda_{m,1}\lambda_{m,2}\lambda_{m,3}\lambda_{m,4} \\
\alpha_1 &= \lambda_{m,1}\lambda_{m,2}\lambda_{m,3} + \lambda_{m,1}\lambda_{m,2}\lambda_{m,4} + \lambda_{m,1}\lambda_{m,3}\lambda_{m,4} + \lambda_{m,2}\lambda_{m,3}\lambda_{m,4} \\
\alpha_2 &= \lambda_{m,1}\lambda_{m,2} + \lambda_{m,1}\lambda_{m,3} + \lambda_{m,1}\lambda_{m,4} + \lambda_{m,2}\lambda_{m,3} + \lambda_{m,2}\lambda_{m,4} + \lambda_{m,3}\lambda_{m,4} \\
\alpha_3 &= \lambda_{m,1} + \lambda_{m,2} + \lambda_{m,3} + \lambda_{m,4}
\end{aligned}$$

Control Law Design

Finally comes the *control law* which is designed, as shown in section 2.1.3, by defining the matrices K, Q and M. The first one is computed by placing the eigenvalues λ_c, i on the matrix $A_c - B_c K$ and it is worth to note that this problem has the same solution of the pole placement proposed in the reference generator design, this means that the components of $K = \begin{bmatrix} k_1 & k_2 \end{bmatrix}$ can be computed using the same formulae exposed before, simply replacing $\lambda_{ref,i}$ with $\lambda_{c,i}$. Then, by solving the equations

$$\begin{bmatrix} H_c + Q\, A_d \\ 0 \end{bmatrix} = \begin{bmatrix} A_c & B_c \\ F_c & 0 \end{bmatrix} \begin{bmatrix} Q \\ M \end{bmatrix}$$

with $Q = \begin{bmatrix} q_0 \\ q_1 \end{bmatrix}$ and $M = m$ as unknowns, it results

$$q_0 = 0; \quad q_1 = 0; \quad m = \frac{h_{c,2}}{b_{c,2}}.$$

2.2.1 Case Study Simulations

The simulation is based on a Matlab-Simulink script where the parameters in the table 2.1 are considered. The simulation time is set to $6\,s$ and the sample time is $T = 5\,ms$.

System Parameters							
Arm Part						Motor Part	
$m\,[kg]$	$l\,[m]$	$r\,[m]$	$J_a\,[kg\,m^2]$	$\beta_a[kg\,m\,s^{-1}]$	τ	$J_m\,[kg\,m^2]$	$\beta_m\,[kg\,m\,s^{-1}]$
5	2	1	6.667	0.5	150	0.02	0.6

Table 2.1: Fine Model Parameters

In addition, as can be seen from the previous sections, there are three different pole placements and for each one of them the respective eigenvalues are defined in **discrete time**. For the sake of simplicity the poles are considered to be real and coincident for all the pole placement problems, and their values are set as follows:

$$\begin{aligned} \lambda_{ref} &= 0.98 \\ \lambda_c &= 0.8187 \\ \lambda_m &= 0.85 \end{aligned}$$

The simulation consists in bringing the arm from the initial position $\theta = 0$ to the position $\theta = \pi$, thus the reference $r(k)$ is a step which starts at time $1\,s$ with

amplitude $r_\infty = \pi$. In following some simulations will be shown, each one for a different parameter error, with the goal to show the robustness of the EMC methodology. The performances are evaluated by means of both the *tracking error* $\underline{e} = \underline{y} - y$ and the *model error* $\bar{e} = y - y_m$, in addition each chart compares the controller performance with noise estimation and the one without it.

SIMULATION 1: $\beta_{eq,m} = \beta_{eq}$, $J_{eq,m} = J_{eq}$ $\rightarrow$ *no parameter errors*

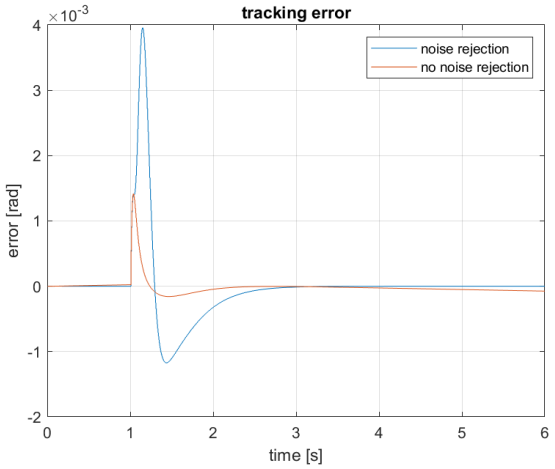


Figure 2.5: simulation 1, $\underline{e}$

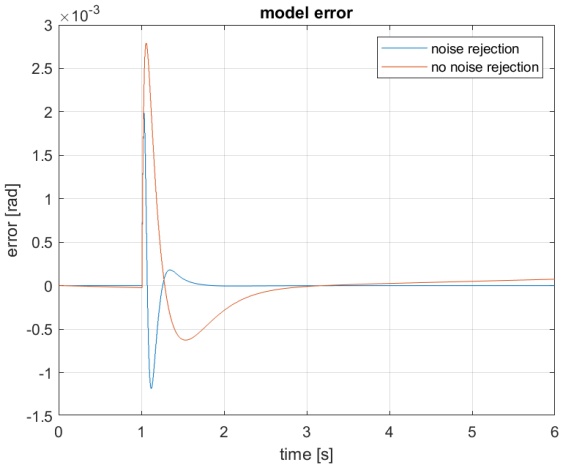


Figure 2.6: simulation 1, $\bar{e}$

Note that in figure 2.5, in terms of magnitude, the red line is even better than blue one, but the problem is that both the errors will converge to a non zero value because of the gravitational force, which slowly pushes down the arm. Meanwhile, with the noise rejection the control action compensates the torque due to gravitational action keeping, as can be seen, the blue lines permanently to 0.

SIMULATION 2: $\beta_{eq,m} = 1.05\,\beta_{eq},\ J_{eq,m} = 1.05\,J_{eq} \ \rightarrow\ 5\%\ of\ error$

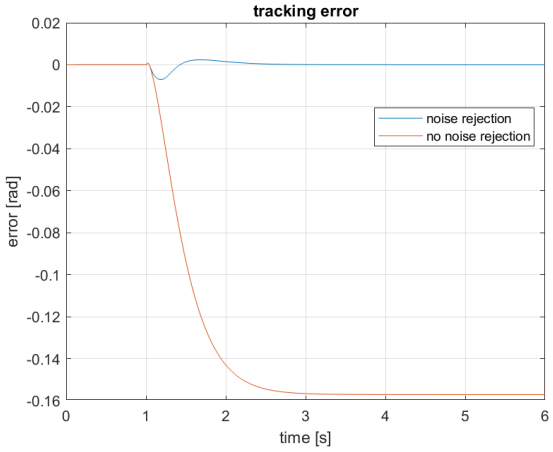


Figure 2.7: simulation 2, $\underline{e}$

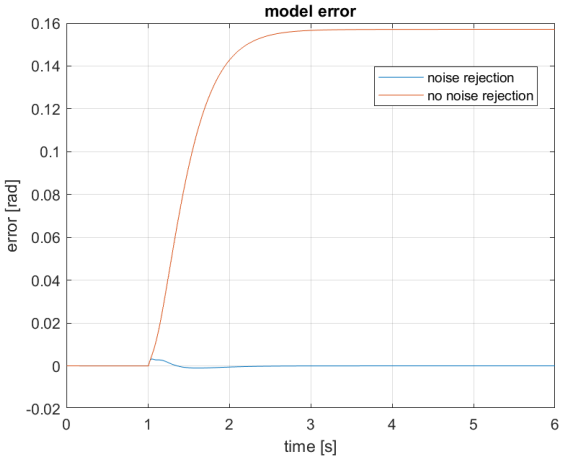


Figure 2.8: simulation 2, $\bar{e}$

Here can be seen how the parameter errors completely influence the performances of the controller without noise estimation, while the errors of the simulations with noise estimator are more or less of the same order magnitude of the ones of simulation 1.

SIMULATION 3: $\beta_{eq,m} = 1.1\,\beta_{eq},\ J_{eq,m} = 1.1\,J_{eq} \ \rightarrow\ 10\%\ of\ error$

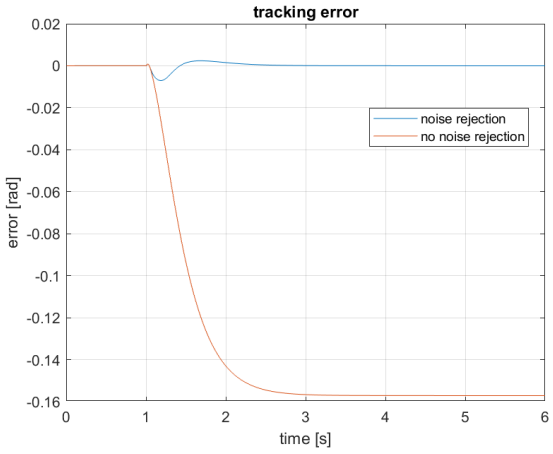


Figure 2.9: simulation 3, $\underline{e}$

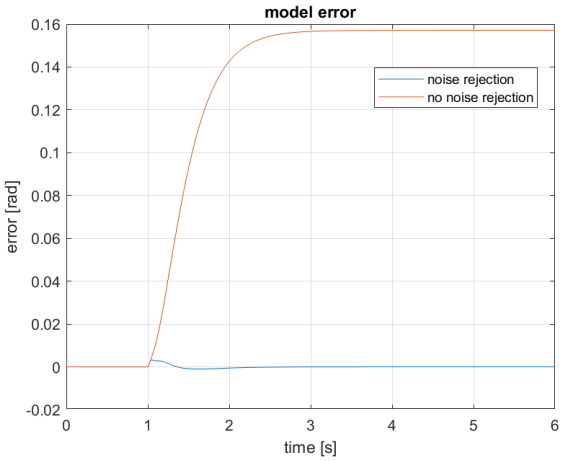


Figure 2.10: simulation 3, $\bar{e}$

Finally, with the third simulation, it is shown how the errors of the models without noise estimator increase with the parameter errors. On the other hand, in terms of magnitude, both model and tracking errors remain almost unchanged in all the three simulations when the noise estimation is on; but, most important, all of them remain on 0 at steady state.

2.3 Asynchronous EMC

The term "*asynchronous-EMC*" [2] derives from the integration of some techniques in the EMC design methodology that makes it suitable to some particular frameworks characterized by asynchronous measurements and commands. **Networked Control Systems** (**NCS**) are a typical example, since they employ the Network to keep *Plant* and *Control Unit* in communication, introducing several timing problems, crucial for control applications, such as *delay* of transmission and delay variability (i.e. *jitter*); other possible problems of the Network communication, as the packet loss or packet exchange, are not addressed in this thesis. The goal of this section is to show how to extend the EMC design in such a way that the above mentioned issues can be properly handled.
The problem to face in this section dictates that the Control Unit receives the measurements asynchronously from the Plant and each measurement arrival triggers the command computation, which is provided to the Plant asynchronously as well. The asynchronism makes the arrival times t_i to be non-deterministic and, consequently, the intervals $T_i = t_{i+1} - t_i$ are potentially different one each other; a possible representation of this concept is given by the figure below.

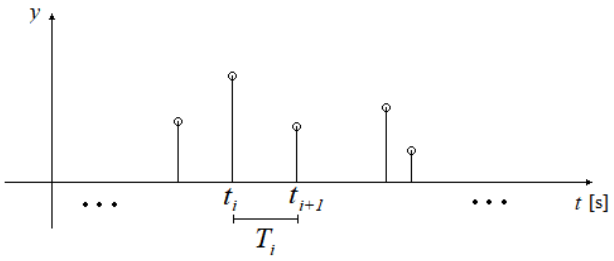


Figure 2.11: sample arrival times with variable intervals

For the construction of the asynchronous-EMC the time-stamp associated to each measurement is needed such that the related time interval T_i, which actually is the variable sampling time, can be derived. As the following sections will better explain, the main idea of the asynchronous-EMC is the adaptability to the variable time interval T_i.

2.3.1 Asynchronous Embedded Model

The first step to construct an asynchronous-EMC is to make the *Embedded Model* adaptive to the variable sampling time T_i. Let's suppose to have a LTI continuous time state equations as the ones represented by 2.1, then it is possible to obtain the related discrete time equations by applying the forward Euler method 2.2, which allows to obtain the following result

$$\begin{aligned}\mathbf{x}(t_i+T_i) &= (T_iA_{CT}+I_{n_x\times n_x})\mathbf{x}(t_i)+T_iB_{CT}\mathbf{u}(t_i)+T_iG_{CT}\mathbf{w}(t_i) = \\ &= A_i\mathbf{x}(t_i)+B_i\mathbf{u}(t_i)+G_i\mathbf{w}(t_i), \\ \mathbf{x}(t_i) &= \begin{bmatrix}\mathbf{x}_c\\ \mathbf{x}_d\end{bmatrix}(t_i), \qquad \mathbf{x}(0)=\mathbf{x}_0;\end{aligned} \tag{2.31}$$

where

$$A_i=\begin{bmatrix}A_{ci} & H_{ci}\\ 0_{n_d\times n_c} & A_{di}\end{bmatrix}, \quad B=\begin{bmatrix}B_{ci}\\ 0_{n_d\times n_u}\end{bmatrix}, \quad G=\begin{bmatrix}G_{ci}\\ G_{di}\end{bmatrix}, \tag{2.32}$$

as you can see the matrices A_i, B_i and G_i adapt to the time interval T_i. In addition it is possible to define the two model outputs $\mathbf{z}_m(k)=F\mathbf{x}(k)$ and $\mathbf{y}_m(k)=C\mathbf{x}(k)$, where, for simplicity, will be considered $F=C$.
At this point, following the guidelines of section 2.1.1 and using the just mentioned adaptive matrices, it is quite simple to write both *actual* and *reference* dynamics of the asynchronous-EM. Indeed for the *actual dynamics* it is possible to the define its two parts as:

- *Controllable Dynamics*

$$\begin{aligned}&\mathbf{x}_c(t_i+T_i)=A_{ci}\mathbf{x}_c(t_i)+B_{ci}\mathbf{u}(t_i)+\mathbf{d}(t_i), \qquad \mathbf{x}_c(0)=\mathbf{x}_{c0}\\ &\mathbf{y}_m(t_i)=C_c\mathbf{x}_c(t_i)\end{aligned} \tag{2.33}$$

- *Disturbance Dynamics*

$$\begin{aligned}&\mathbf{x}_d(t_i+T_i)=A_{di}\mathbf{x}_d(t_i)+G_{di}\mathbf{w}(t_i), \qquad \mathbf{x}_d(0)=0\\ &\mathbf{d}(t_i)=H_{ci}\mathbf{x}_d(t_i)+G_{ci}\mathbf{w}(t_i)\ \ ,\end{aligned} \tag{2.34}$$

while the *reference dynamics* can be defined by the following equations:

$$\begin{aligned}&\underline{\mathbf{x}}(t_i+T_i)=A_{ci}\,\underline{\mathbf{x}}(t_i)+B_{ci}\,\underline{\mathbf{u}}(t_i), \quad \underline{\mathbf{x}}(0)=\underline{\mathbf{x}}_0\\ &\underline{\mathbf{y}}(t_i)=C_c\,\underline{\mathbf{x}}(t_i)\ \ .\end{aligned} \tag{2.35}$$

Notice that the state equations matrices can't be computed once and for all, indeed it is needed to compute them at each sample arrival but this, in terms of computational time, is not a problem, as it is a relatively low number of operations.

2.3.2 Asynchronous Noise Estimator and Control Law

As previously mentioned, the main characteristic of the asynchronous-EMC is its adaptability to the variable time interval T_i, not only by changing the discrete time model with T_i (as seen in 2.3.1), but also by adjusting the response speed to the time elapsed between two samples. The elements which allow to perform these adjustments are the eigenvalues of both the *observer* and the *controller*, which must vary with T_i as $\lambda^{DT} = e^{\lambda^{CT} T_i}$. Therefore at design time the eigenvalues are chosen in continuous time and, in order to keep the desired frequency behaviors of both the observer and the controller, it is needed to dynamically compute the respective discrete time eigenvalues, where the desired frequency behavior for the controller response is imposed by the plant bandwidth, while for the noise estimator it is taken by the bandwidth of the dynamics to estimate. These bandwidths force some constraints over the sampling time T_i since a too high value can cause a too fast response which can be cut, while a too low value of T_i involves a too slow response which doesn't let to follow the dynamics of interest: in the worst case, it can cause instability.
Established this, let the assumptions over the state equations matrices enunciated in 2.1.1 and 2.1.2 to hold in this case too, then it is possible to construct both the *noise estimator* and *control law* exactly as it is done in the canonical EMC.
For the **noise estimator**, if a *dynamic estimator* (see 2.13) is designed then it holds $A_{mi} = \begin{bmatrix} A_i - G_i L_i C_i & G_i M_{mi} \\ -C_i & 1 - \beta_i \end{bmatrix}$, otherwise (see 2.10) the matrix A_{mi} can be simply written as $A_{mi} = A_i - G_i L_i C_i$, where L_i, and eventually M_{mi} and β_i, can be computed by placing the eigenvalues $\lambda(A_{mi})$.
On the other side, the design of the **control law** (see 2.19) needs the following two conditions to hold:

$$1.\ |\lambda_k(A_{ci} - B_{ci} K_i)| < 1, \forall\ k = 1, ..., n_c\,, \tag{2.36}$$

$$2.\ \begin{bmatrix} H_{ci} + Q_i\, A_{di} \\ 0 \end{bmatrix} = \begin{bmatrix} A_{ci} & B_{ci} \\ F_{ci} & 0 \end{bmatrix} \begin{bmatrix} Q_i \\ M_i \end{bmatrix}\ . \tag{2.37}$$

By solving the second one, it is possible to compute the two matrices M_i and Q_i, while matrix K_i is the solution of the pole placement problem which guarantees the achievement of the first condition. Therefore it is clear that, in practice, what is changed is that the unknowns must be computed at each time step because of the variability of both the state space matrices and the discrete time poles.

2.4 Asynchronous EMC Applied to a Mechanical Arm Driven by DC Motor

The case study analyzed in this section is the same treated in 2.2, hence the overall design of the EMC won't change so much. The additional assumption introduced in this problem which make sense the usage of an asynchronous-EMC is the fact that the *extended plant* is remotely controlled by the *control unit*, making the system to assume the structure of an NCS, which involves that the EMC must deal with both the *transmission delay*, denoted with t_d, and the *variable sampling time*.

Extended Plant Definition

Recall that the plant consists of a mechanical arm rotating on a vertical plane (thus subject to disturbance toques due to gravity force) where the command input $u(t)$ is the motor torque, transmitted to the arm by means of a gearbox with ratio τ ; the scheme in figure 2.2 can be taken as a reference for this extended plant structure once again, while the parameters of the mechanical arm system can be taken from the table 2.1.

Embedded Model Design

The model in continuous time domain is represented by the 2.26, where the nonlinearity provided by the gravity action is treated as a causal disturbance dynamic. By applying the forward Euler method, as explained in theoretical part, it is possible to obtain the time variable DT matrices:

$$A_i = \begin{bmatrix} 1 & T_i & 0 \\ 0 & -T_i \dfrac{\beta_{eq,m}}{J_{eq,m}} + 1 & T_i \\ 0 & 0 & 1 \end{bmatrix}, \quad B_i = \begin{bmatrix} 0 \\ T_i \dfrac{1}{\tau J_{eq,m}} \\ 0 \end{bmatrix}, \quad G_i = \begin{bmatrix} 0 & 0 \\ T_i & 0 \\ 0 & T_i \end{bmatrix}, \tag{2.38}$$

$$n_c = 2, \;\; n_d = 1, \;\; n_u = 1, \;\; n_w = 2, \;\; n_y = 1.$$

At this point it is quite simple to write the *Actual Dynamics* for this model, indeed it suffices to apply the formulae 2.33 and 2.34 as seen before.
For the *Reference Generator* it is used the same idea of the EMC case study previously analyzed, namely a static state feedback control (figure 2.3) which provides a smoother reference to the system. Therefore, using the just shown matrices, it is possible to construct the *reference dynamics* using 2.35 and then to establish the value of $K_{ref,i}$ by placing the eigenvalues $\lambda(A_i - B_i K_{ref,i})$. Note that, even in this case, it is possible to dynamically change the desired DT eigenvalues starting from the ones in CT domain, defined as λ_{ref}^{CT}.

Noise Estimator Design

Since a *dynamic* noise estimator is implemented, indeed it holds $n_w < n$, taking as reference the matrices shown above, it is possible to define the observer closed loop state matrix as:

$$A_{m,i} = \begin{bmatrix} 1 & T_i & 0 & 0 \\ -T_i\, l_{0i} & -T_i \dfrac{\beta_{eq,m}}{J_{eq,m}} + 1 & T_i & T_i\, m_{0i} \\ -T_i\, l_{1i} & 0 & 1 & T_i\, m_{1i} \\ -1 & 0 & 0 & 1 - \beta_i \end{bmatrix}$$

and then, by imposing the set $\lambda(A_{m,i})$ at the desired one, it is possible to compute the unknowns l_{0i}, $l_{1,i}$, $m_{0,i}$, $m_{1,i}$ and β_i in function of the eigenvalues. Note that the problem is exactly the same exposed in the previous case study, thus, posing $m_{0i} = 0 \ \forall \ i$, a solution similar to 2.30 can be applied in this case too, with the difference that the unknowns must be computed at each time step. At this point, having both $L_i = \begin{bmatrix} l_{0i} \\ l_{1i} \end{bmatrix}$ and $M_{mi} = \begin{bmatrix} m_{0i} \\ m_{1i} \end{bmatrix}$, it is possible to build the already known dynamical estimator of the driving noise:

$$\bar{\mathbf{w}}(t_i) = M_m\, q(t_i) + L\, \bar{e}(t_i) \ ,$$

where

$$q(t_i + T_i) = (1 - \beta_i)\, q(t_i) + \bar{e}(t_i) \ , \quad q(0) = 0 \ .$$

Control Law Design

For the *control law* the strategy is identical to the one of the previous case study, indeed it is possible to compute very simply Q_i and M_i by solving the Devison-Francis condition 2.37, which provides:

$$Q_i = \begin{bmatrix} 0 \\ 0 \end{bmatrix}; \ \ M_i = \frac{h_{ci,2}}{b_{ci,2}}$$

and then it is possible to compute K_i such that $\lambda_1(A_{ci} - B_{ci}K_i) = \lambda_{ci,1}$ and $\lambda_2(A_{ci} - B_{ci}K_i) = \lambda_{ci,2}$, where $\lambda_{ci,1}$ and $\lambda_{ci,2}$ are the desired DT eigenvalues.

2.4.1 Simulations Results

The data exposed in this section are the results of simulations performed in the framework MATLAB-SIMULINK, where the extended plant is built using the parameters in the table 2.1, while the EM parameters are affected by 5 % error, as

done in simulation 2 of section 2.2.1. The simulation lasts $t_f = 5\,s$ and the sampling time for the plant output is $T = 5\,ms$. As already said, the eigenvalues which must be placed in the three pole placement problems, must be defined in continuous time. In this case, for the sake of simplicity, the eigenvalues are considered to be real and coincident for all the problems and are:

$$\lambda_{ref,1}^{CT} = \lambda_{ref,2}^{CT} = \lambda_{ref}^{CT} = -3\,, \quad \lambda_{c,1}^{CT} = \lambda_{c,2}^{CT} = \lambda_{c}^{CT} = -20\,,$$
$$\lambda_{m,j}^{CT} = \lambda_{m}^{CT} = -15 \quad for \quad j = 1, ...,4\,.$$

Furthermore two artifice are used such that a NCS can be emulated in the simulation:

- the '*Transport Delay*' Simulink block, connecting control unit and extended plant in both directions, which retards the signals with a delay of $10\,ms$.
- the '*Triggered Subsystem*' which is a particular Simulink subsystem executed at each rise front of a square wave with a period which varies uniformly from $2\,ms$ to $30\,ms$.

With this configuration it happens that t_d is within the interval $[12, 40]\,ms$ and, consequently, $T_i \in [2,\ 30]\,ms$ with a uniform distribution; the figure 2.12 shows what has just been said, indeed it represents the behavior of the sampling times in the first 100 samples.

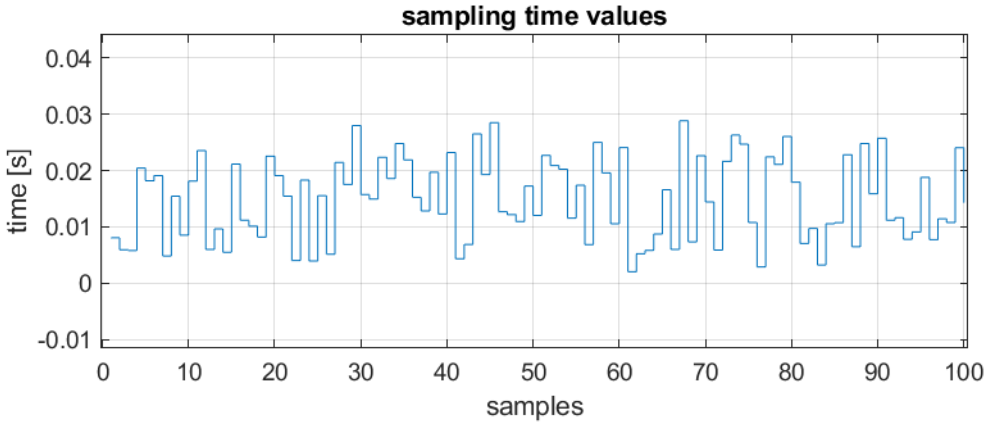


Figure 2.12: Variability of the Sampling Times

The following results show the response of the controlled system to a step reference with a rise front at $0.5\,s$ from the simulation start; they are analyzed by means of plant and model outputs, respectively y and y_m, compared to the generated reference $\underline{y}$ (figure 2.13), and the model and tracking errors, respectively $\bar{e}$ and $\underline{e}$ (figure 2.14). The two figures below (second one in a more detailed fashion) show

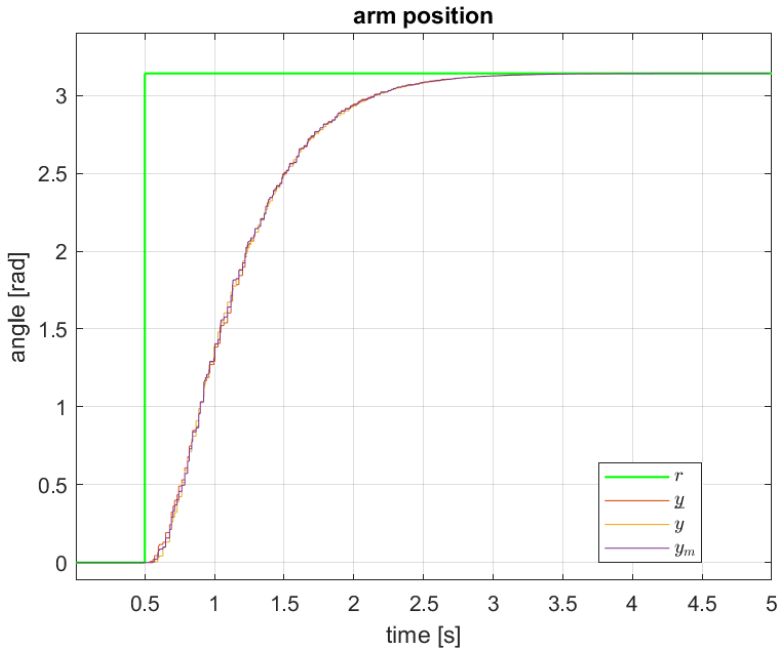


Figure 2.13: model and plant outputs tracking the reference (green) following the trajectory of y (red)

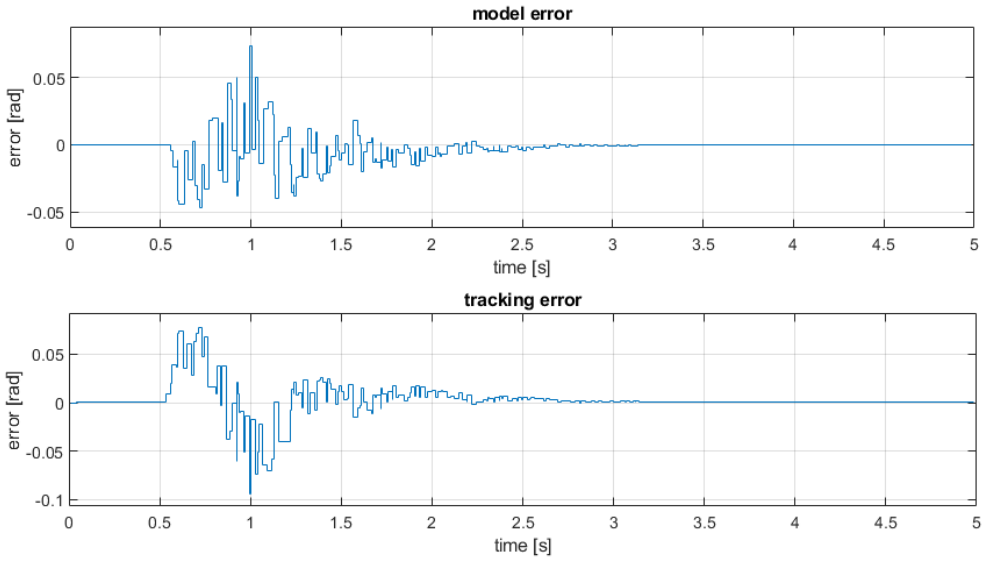


Figure 2.14: model (top) and tracking (bottom) errors

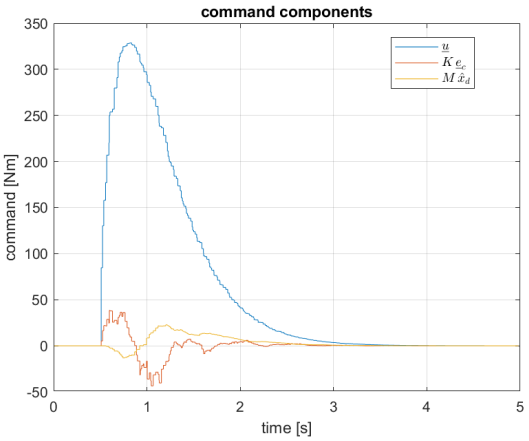


Figure 2.15: reference generator output command (blue line), tracking error component (red line) and disturbance rejection component (yellow line)

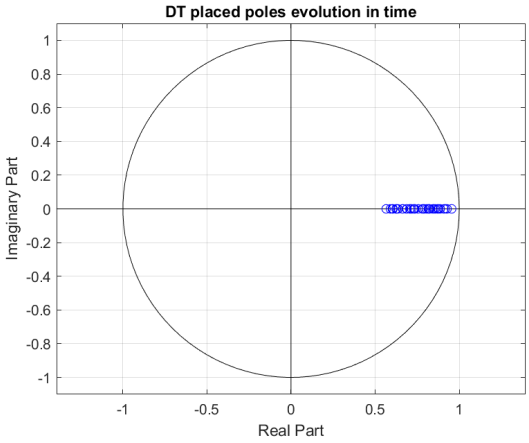


Figure 2.16: state observer discrete time poles evolution

the good behavior of the asynchronous-EMC, which lets the output to track very precisely the reference.

The figure 2.15 shows the three components of the command u, including the one represented by yellow line which allows to reject the disturbances enclosed in the state $\hat{x}_d$.

Another important aspect of the asynchronous-EMC is the variability of the DT eigenvalues which, as told in the theoretical part, guarantees to keep the desired frequency behavior even when the sampling time changes at each time step. The figure 2.16 shows the variation, in the z-plain, of the DT poles placed in the extended state observer, each blue circle represents one pole and all of them are stable poles since lay within the unit circle.

Chapter 3

Network Architectures

As anticipated in the previous chapters, the goal in this thesis is to test the effectiveness of the asynchronous-EMC applied on a real NCS. To this aim, it is necessary to build up a network structure between the plant and the control unit, which are respectively represented by a Raspberry Pi board, which is attached to a robot, and the author's laptop, where the controller is implemented. Two different solutions are explored for the establishment of this communication: the first one includes the usage of an *application programming interface* (API) named **pcap**, and useful for capturing raw packets; the second solution contemplates a traditional **socket** employment, more in detail a UDP transport protocol is used for this project.

3.1 Packet Capture Libraries

The name pcap stands for *packet capture*, in fact the main purpose of this API is to obtain, at application level, all the packets read by a network interface in a raw format, bypassing the whole protocol stack; for this reason it is at the basis of the main traffic analysis softwares such as *Wireshark* or *tcpdump*.
The libraries which allow to access to pcap functionalities are *libpcap* and *WinPcap* (both written in `C`), the first one is the original library for Unix while the second is a porting of libpcap for Windows operating systems. Note that the client side, represented by the robot system, must be implemented in a Linux environment, thus libpcap is needed, while the server side is on Windows and it needs the WinPcap version. Before talking about the functions of these libraries, a brief overview of the pcap design is provided in order to understand all the critical and beneficial aspects of their employment.

3.1.1 Libpcap and WinPcap architectures

Without entering in implementation details [8], the basic ideas of *libpcap* and *WinPcap* libraries are the same and are focused on two main actions:

- **filtering** the packets coming from the network interface such that the only packets arriving to the user application are the ones that achieve certain constraints;
- **buffering** data so that the number of readings (or system calls, if the buffer is in kernel space) can be reduced;

the strength of this capture system stands in performing these two actions at kernel space, since it allows to significantly reduce the number of transmitted packets from the network interface to the user space.
The filtering process starts at user level where the library allows to define some text-based commands (with a specific syntax) which must be compiled such that the resulting code can be sent to the filtering machine located in kernel. Doing so, each arrived packet can be processed by the kernel and all the packets which don't satisfy the user-defined constraints are discarded, the others are sent to the kernel buffers. In fact, before sending them to user space, it is fundamental to preliminarily store packets in kernel space to perform as few system calls as possible, since they are critical in terms of performance; after that it is possible to fill user-space buffer with a single *read()*.

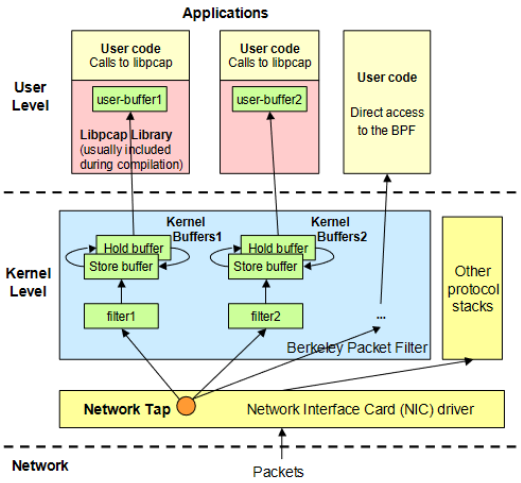


Figure 3.1: libpcap architecture

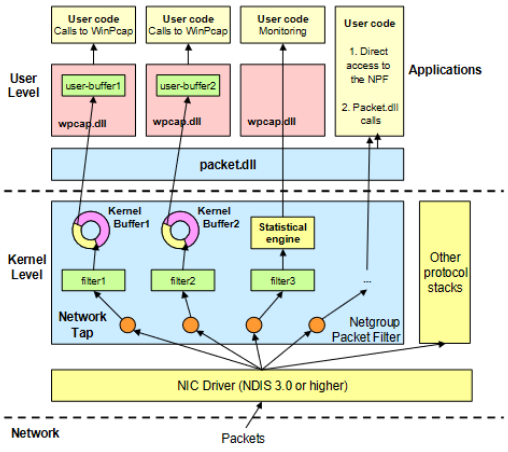


Figure 3.2: WinPcap architecture

Figures 3.1 and 3.2 (taken by [8]) show respectively the libpcap and WinPcap architectures, highlighting some differences such as the kernel buffers implementations or the presence of additional elements at WinPcap user level. Anyway the goal here is not to delve into these details, but it is important to point out that both the architectures need the support of a component at kernel level which is able to communicate with the libraries. For WinPcap this is not a problem since the installation of the library includes a driver called Netgroup Packet Filter (NPF) which operates at kernel level (see fig. 3.2), but for libpcap the Barkeley Packet Filter (see fig. 3.1) is not present in all the Unix operating systems, in this case both filtering and buffering actions must be emulated at the user level with the consequent loss of pcap benefits.
Once these aspects have been clarified, it becomes easier to understand the strengths and weaknesses of a communication based on pcap. This solution provides high performances since the packets are processed only from the filter, bypassing the protocol stack; this allows to save time in packets processing with respect to IP-based network communications. On the other hand the performances are strongly influenced by the buffering mechanism and, as will be shown in the next sections, the management of buffering by means of parameters defined by the code developer becomes crucial.

3.1.2 Libpcap functions

At this point a brief overview of functions used in this project and exposed by both *libpcap* and *WinPcap* must be provided. The first mandatory step to perform a packet capture is the choice of the network interface which will provide the packets.

```
int  pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf);
void pcap_freealldevs(pcap_if_t *alldevs);
```

The two functions above are used respectively to obtain the list of all the devices, passing a list pointer as reference, and to free the memory occupied by the list itself; once the device is chosen it must be opened. The opening action is performed by

```
pcap_t* pcap_open_live(const char* device, int snaplen, int promisc,
                       int to_ms, char* ebuf);
```

which opens the device identified by ***device*** argument, located in the structure ***pcap_if_t***, and sets the capture behavior by means of: ***snaplen***, which define the maximum number of bytes to capture, ***promisc***, which specifies if the interface must be set to promiscuous mode and ***to_ms***, which represents the read timeout

in milliseconds. About this last parameter, it is worth to notice that it represents the time to wait before returning from reading, regardless of whether packets have arrived in the meantime; this means that, for the purposes of this project, it can be a crucial parameter since choosing a too high value would risk to have too many packets buffered together and, therefore, to have too many consecutive arrivals with time intervals ≈ 0. On the other hand, by choosing it too low, a performance drop could be experimented caused by too frequent context switching. The last argument, namely *ebuf*, is used to obtain an eventual error description. Finally this function returns a pointer to the structure *pcap_t*, which is the identifier of packet capture process.
Before start reading it is needed to set the filter for the obtained pcap identifier, this is made by the following functions

```
int pcap_compile(pcap_t* p, struct bpf_program* fp, char* str, int optimize,
                 bpf_u_int32 netmask);
int pcap_setfilter(pcap_t* p, struct bpf_program* fp);
```

where the first one allows to compile the textual filter command written in *str* argument, by saving the resulting code in the space pointed by *fp*. Notice that the function also requires the identifier of the capture and the netmask of the network in which the interface is. After that the compiled code must be sent to the filter machine by means of the second function, which requires the structure where filter code is saved.
At this point it is possible to start the capture, with the certainty that all the readings are triggered only by packets which achieve the constraints specified in *str* parameter.

```
int pcap_loop(pcap_t* p, int cnt, pcap_handler callback, u_char* user);
```

The function above starts the capture and blocks the thread which calls it in an event loop, until the number of read packets reaches the value indicated in *cnt* or until a call of the following function

```
int pcap_breakloop(pcap_t* p);
```

that must be performed by the thread which is in the loop; in the meanwhile at each packet arrival the function pointed by *callback* is called. In particular, the function pointer type is defined in the following way

```
typedef void(*pcap_handler)(u_char* user, const struct pcap_pkthdr* pkt_header,
                            const u_char* pkt_data)
```

where the data of each packet is stored in the location pointed by ***pkt_data***, while the structure pointed by ***pkt_header*** simply encapsulates an header related to the capture which must not be associated to protocol headers.
Anyway it is possible to use another method to read packets which bypasses the callback mechanism and can be actuated by calling the function

```
int pcap_next_ex(pcap_t* p, struct pcap_pkthdr** pkt_header,
                 const u_char** pkt_data);
```

which returns when one of two conditions are verified: (i) a packet is ready to read or (ii) ***to_ms*** milliseconds are elapsed from its call. The advantage of this function is that, using the return value of this function, it is possible to implement a loop with customized exit conditions. In the project, this reading method is used on server side, while client side uses the first one.
Finally, the last function that will be shown in this section is used for sending packets. Anyway, as for the readings, it deals with raw packets which means that if the intention is to communicate within an IP infrastructure (passing through IP routers) then the whole packet with appropriate headers must be built. It goes without saying that such a solution would not be convenient in terms of implementation, so it is considered an acceptable solution for a first phase in which the robot and the PC are connected through a p2p connection. The cited function is the following one

```
int pcap_sendpacket(pcap_t* p, u_char* buf, int size);
```

and it requires both the pointer to the buffer, where data must be picked up, and the size of data to send.

3.1.3 Performances Evaluation: p2p connection

The setting that is going to be evaluated in this section is a p2p connection via Ethernet cable between laptop and Raspberry Pi attached to the robot. In this condition two performance indices are considered:

- *Round Trip Time* (RTT), in order to understand the magnitude of the transmission delays;

- *RTT variance*, that is needed to obtain a good representation of the transport delay variation, which is a fundamental parameter for the application of a remote control.

The first performance index is obtained by means of a simple client-server program with which the laptop sends some 'dump' packets to the Raspberry, which replies sending back the same packets it reads. Two timestamps are saved: the first one when the packet is sent by the laptop, the second one when it returns, their difference gives the RTT of the packet. This procedure is performed for $N = 1000$ packets. More in detail, the transmitted packets are 92 Bytes each and are sent by the client each $100\,ms$. As said in previous sections, pcap is able to manipulate raw packets, which means that it is possible to send and receive packets without any protocol header except for that of the Ethernet protocol, since the network interface must be able to recognise MAC addresses. Therefore the packet sent in this test has the format represented in figure 3.3. As can be seen, Ethernet header

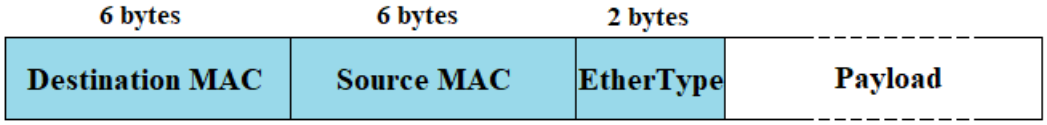


Figure 3.3: packet format

is present in the packet but payload field do not contain any other protocol header and it is filled only with 0 bits for this application. In addition, in order to perform a strong filter, the *'EtherType'* field is set to the unused value 0x8000, with the aim to construct a filter based on this field, guaranteeing that the captured packets, in both the client and the server sides, are only the ones transmitted for this test. It is worth to notice that the RTT includes also the time which the packet spends to pass from the interface to the user application and vice versa. Figure 3.4 shows the RTT trend for $N = 1000$ samples, and three horizontal lines which represent the mean value (in red) and the margins given by standard deviation (in green); approximating this behavior to a Gaussian distribution, it is possible to assume, from the theory of the statistics, that about 68% of the samples should fall within the green lines and the experiment result show that even more than 680 sample lie there. The statistical computations of mean value and standard deviation are

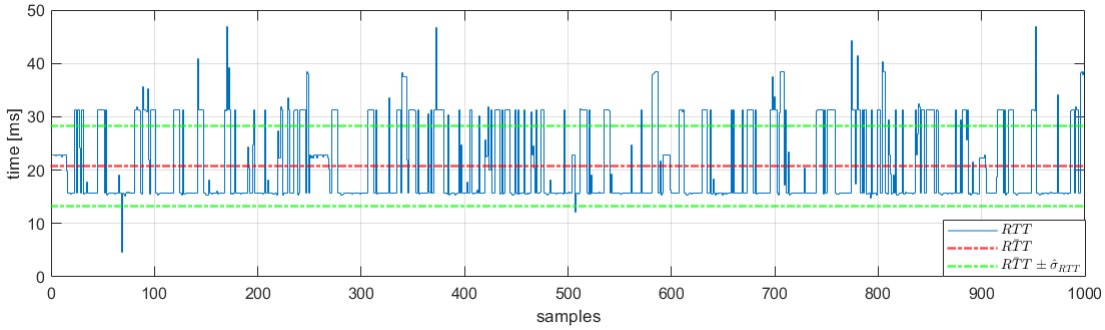


Figure 3.4: test result: Round Trip Time values

performed as follows:

$$\bar{RTT} = \frac{1}{N}\sum_{i=1}^{N} RTT_i = 20.7573\,ms, \quad \hat{\sigma}_{RTT} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(RTT_i - \bar{RTT})^2} = 7.5017\,ms$$

allowing to obtain an indirect evaluation of the transmission delay for this Network architecture:

$$\bar{t}_d = \frac{\bar{RTT}}{2} = 10.3787\,ms \approx 10.5\,ms\,.$$

In addition it is very important to notice that the standard deviation measurement is fundamental for the aims of this application since the variability of the sampling time is crucial for the Asynchronous-EMC and, although $\hat{\sigma}_{RTT}$ is not a direct measurement of sampling time variability, it is still a good index for it. In conclusion, this experiment was performed to understand the network response in the p2p setup described above, trying not to influence the data with the buffering mechanism, which would influence the results if the packets would be transmitted too fast.

3.2 UDP Sockets

The second, and more classic, solution requires the use of UDP sockets for both robot and server sides. Sockets are software objects that provide an inter-process communication (IPC) solution between processes running either on the same machine or on dislocated machines which can reach one each other by means of Network infrastructure. Typically three kinds of sockets are provided to the developer:

- **Datagram** sockets, which allow to send single messages to the *specified* endpoint;

- **Stream** sockets, which implement connection-oriented communication that allow to send a *flow* of messages;

- **Raw** sockets, which are used for particular protocols;

in particular, when it comes to Network sockets, the above cited first two types refer respectively to UDP/IP sockets and TCP/IP ones. Both of them need to be bound to specified *IP address* and UDP or TCP *port*, this action allows the sockets to receive all the packets arriving to the specified port-address couple; in addition only TCP sockets need to send an explicit requirement of opening the connection to an endpoint which must be waiting for a proper connection query. Finally, for the packet handling, *read* and *write* actions can be performed on socket objects according to a FIFO order.
Among the various tools for socket programming, the ones chosen for this thesis are: Unix native sockets, for robot side, and the sockets of the framework `Qt`, for the Windows environment of the server. Furthermore UDP protocol is chosen for this application since, even if it can't avoid packet loss, it still provides a more efficient solution than TCP protocol which is heavier due to flow and congestion control.

3.2.1 Unix native sockets libraries

In this section some of the used functions of Unix socket API are provided, the information related to the meaning of each symbol come from online Linux manual [9]. They are exposed in the source file "*sys/socket.h*", and it will be shown that some of the peculiarities of the sockets summarized in the previous section will be present in this implementation too.
The first function that must be presented is the one which is in charge of build the socket, and is declared as shown below.

```
int socket(int domain, int type, int protocol);
```

The return value of this function is the file descriptor which refers to the created socket, while the three arguments determine what kind of socket it must be built. The first parameter, named ***domain*** specifies the type of communication it must be established with the socket, some possible values for that parameter are: `AF_UNIX`, for a IPC between processes in the same device, `AF_INET` and `AF_INET6` respectively for IPv4 and IPv6 Network communication or `AF_BLUETOOTH` for bluetooth links. The argument ***type*** is the one which allows to select the kind of socket, choosing between some defined types such as: `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW` (see 3.2) and others more. Finally ***protocol*** specifies the communication protocol for

the created socket, even if in the most of the cases only one protocol is implemented for each socket type, hence this parameter is usually set to 0.
For our aims the needed socket is a UDP/IP one, this means that three parameters are set respectively to: `AF_INET`, `SOCK_DGRAM` and 0.
Once constructed the socket it is possible to bind it to given address and port, this can be done by means of the function *bind()* declared as follows.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

First of all it can be noticed that the first parameter is the identifier of the socket (i.e. the returned value of *socket()*) while the following ones are needed to specify the address and the port to bind. In particular the structure ***sockaddr*** contains the address, the port and an identifier of the address family (for IPv4 it is `AF_INET`), while the third argument simply is the length of the address structure; note that this last argument is needed because the address length can vary from one address family to an other.
The last function of Unix sockets used in this project is the one which allows to send packets, shown in the code below.

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

This is one of the three functions for sending packets and is mainly used for Datagram type sockets. As can be seen the first parameter is, once again, the socket identifier, while second and third parameters are respectively the pointer to the buffer containing data to send and its length. In addition, since this kind of socket doesn't expect to open a connection, destination port and address must be provided through the last two arguments using the same procedure already explained for the *bind()* function. Furthermore the ***flags*** argument is used to communicate the sending behavior, two examples are represented by the flags `MSG_DONTWAIT`, which makes the operation to be non-blocking, and `MSG_DONTROUTE`, used to send packets without pass by gateway.
Note that no functions for reading packets are shown in this section, this because on the Raspberry it was possible to choose the interface which uses WiFi protocol to perform a packet capture by means of libpcap functionalities.

3.2.2 QUdpSocket

As anticipated in previous sections, on server side the socket communication is implemented by means of an object provided by `Qt` framework, named `QUdpSocket`.

First it can be appropriate to make a brief digression on `Qt` and on its main characteristics, [10].
`Qt` is a multiplatform library, originally written in `C++`, for the developing of graphical interfaces; it is strictly based on *signaling* paradigm, which is also the strength of this framework since it allows to implement in a very convenient way the model-contol-view (MCV), which is a pattern at the basis of graphical interfaces. However, thanks to this signaling mechanism, `Qt` is extremely suitable for any object that has to handle events coming from the operating system: starting from the button widget, receiving the mouse click event, up to the sockets that must manage the interrupts triggered by packets arrivals. The signaling system is based on two kinds of functions:

- *`slots`*: roughly speaking, they are the callbacks of the signals, in fact they are called whenever a signal to which they are registered is issued;
- *`signals`*: they must only be declared but never defined; their only role is to be emitted (by means of the macro `emit`), so that the slots registered to them can be called back.

The function that allows to construct a signal-slot structure (see Fig. 3.5) is the method named *`connect()`* which typically accepts four arguments: the object that owns the signal, the signal itself, the object that owns the slot which must be attached to the signal and the slot itself. At this point, without deepening the

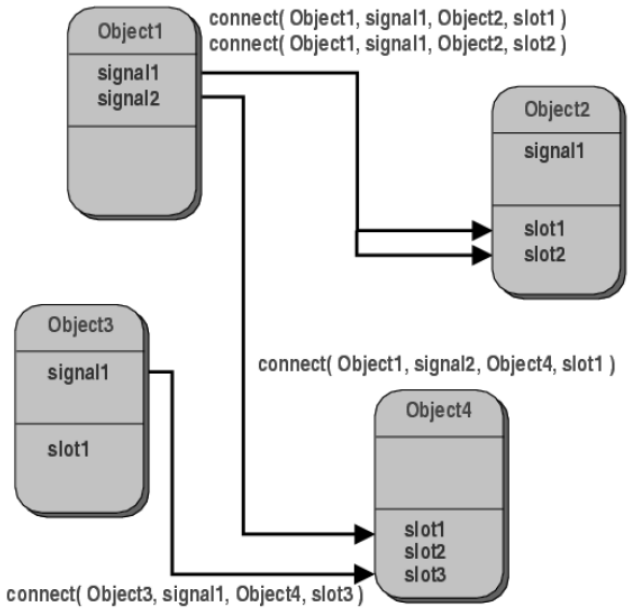


Figure 3.5: signals and slots conceptual architecture

matter of events, for the thesis purposes it is sufficient to know that, at each packet arrival, the QUdpSocket object emits the signal called *QUdpSocket::readyRead()*, to which a slot must be hooked so that it can manage a packet reading. More in detail, the QUdpSocket member functions that have been used in this project are three, one for binding and the others for reading and writing datagram, and are defined as follows.

```
bool QAbstractSocket::bind(const QHostAddress &address, quint16 port=0,
                           QAbstractSocket::BindMode mode=DefaultForPlatform)

qint64 QUdpSocket::readDatagram(char *data, qint64 maxSize,
                            QHostAddress *address=nullptr, quint16 *port=nullptr)

qint64 QUdpSocket::writeDatagram(const char *data, qint64 size,
                                 const QHostAddress &address, quint16 port)
```

First function needs to bind the socket to the specified address-port couple by means of the `QHostAddress` class, which can encapsulate different types of addresses, and the argument ***port***. After binding, the signal *QUdpSocket::readyRead()* is emitted whenever a packet arrives at the specified address and port. At this point, as said before, it suffices to connect a slot which performs a reading of the datagram by means of the second function shown above. It can be seen that the reading method requires a buffer where to save data and the maximum number of bytes to read, it returns the number of effectively read data; it is also possible to obtain the source address and port by means of its last two arguments.
Finally the first two arguments of the writing method are respectively the pointer to the buffer, where lies the data to transmit, and the size of data to transmit; furthermore the other two arguments are used to specify the destination address and port.

3.2.3 Performance Evaluation: UDP connection

As done in section 3.1.3, the aim here is to evaluate the performances of a network infrastructure built by means of UDP connection between two hosts (i.e. the Raspberry and the laptop) between which a router is posed. For this test, a home router connected to both devices via WiFi protocol is used and, in order to ensure that it behaves like a third-layer device rather than a simple switch, the packets exchanged between hosts have the *public* address of the router as their destination addresses so that the packets arrived to the router can be forwarded to the desired hosts. This is possible after a proper *PAT* setting, which is the router component in charge of forwarding packets on the private network according to their destination port. Transmission times are expected to be much more variable than in the

p2p case analyzed above since a third element (the router) is inserted into the communication and it is expected to manage other traffic in addition to the one dedicated to this application.
Also in this case the main performance indices are the RTT and its variance. In order to obtain them a client-server application as the one of section 3.1.3 is built. Thus the laptop sends and receives a packet by means of the object `QUdpSocket`, while the Raspberry sends back the arrived packets. The RTT is computed on for each sent packet, by measuring the time elapsed from the departure of a packet to its arrival.
More in detail $N = 1000$ packets are sent each $100\,ms$ and each one of them is 50 Bytes long, the results of the test are exposed in figure 3.6 which, compared to the figure 3.4, shows a greater variability of RTT but an unexpected lower mean value (red line). In fact the mean value $\bar{RTT}$ and the standard deviation $\hat{\sigma}_{RTT}$ are respectively:

$$\bar{RTT} = 16.8910\,ms, \quad \hat{\sigma}_{RTT} = 20.5035\,ms$$

hence the expected delay time can be considered to be

$$\bar{t}_d = 8.4455\,ms \approx 8.5\,ms\ .$$

It is worth to notice that, in this case, an approximation to a Normal distribution

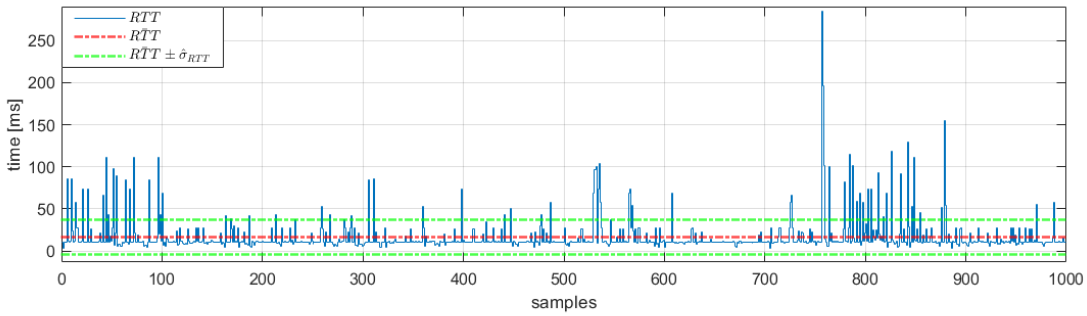


Figure 3.6: RTT values for UDP communication

can result more forced than the previous case since the computation of a standard deviation do not takes into account a lower bound for the RTT, provided by the technological constraints. This problem can be noticed by the bottom green line of figure 3.6 which goes below $0\,s$, covering a zone that is certainly unrealistic for RTT values. In any case it is undeniable that the result of this last test gives very variable RTT values.

As already said both the solutions are experimented for the application of the Asynchronous-EMC, and it is expected to have two results radically different due

to the response of the network infrastructure. In fact in both cases the transport delay is not relevant since it is a time comparable to the reaction times of the robot system, but it is clear that for connection implemented by means of UDP sockets the EMC performances are expected to be more deteriorated compared to the case with a p2p connection, due to the higher variability of the sampling time. In any case the second solution is considered as the one of greater practical validity since it is the solution closer to a real application case.

Chapter 4

GoPiGo3: Mobile Robot

In order to test the theoretical results, in the following sections an experimental application is performed over a two-wheeled differential drive mobile robot; in particular the robot used for these tests is the *GoPiGo3*, which is an educational purpose product of *Dexter Industries*. This robot microcontroller can be attached to the *Raspberry Pi* such that a Raspbian-like OS can be employed in order to manage the I/O devices for a full control of robot movements.

4.1 Hardware Specifications

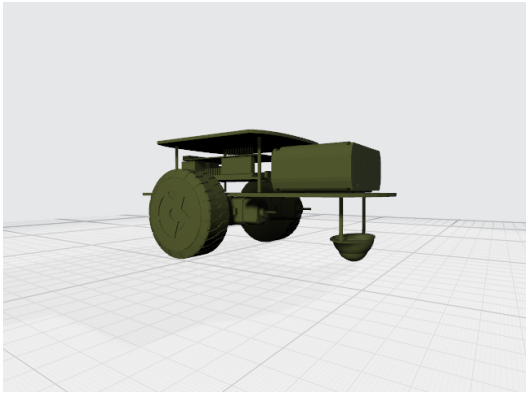


Figure 4.1: back view of a GoPiGo3 3D model

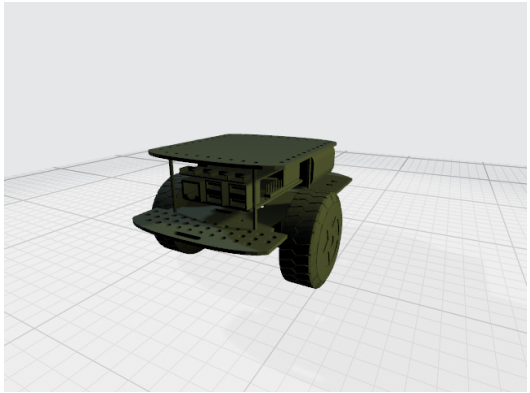


Figure 4.2: front view of a GoPiGo3 3D model

The figures 4.1 and 4.2 show the GoPiGo3 shape reproduced by a 3D model taken from [11]. As said before this is a differential drive mobile robot indeed, as the two pictures show, it has three wheels:

- one rear omnidirectional passive wheel that has the role of stabilizing the robot,

- two active front wheels that are allowed to rotate only about the axis parallel to the floor, for this reason they are said to be "fixed".

Since the wheels can't steer, the chassis rotation about the z axis is allowed by the different speed rotation of the two active wheels, thus it is needed the presence of two motors, one for each front wheel, with a gear ratio of $\tau = 120$. In addition both the motors are endowed of an incremental magnetic encoder with 6 pulse counts per rotation that means to have 720 pulse counts per wheel rotation [12], providing an angular resolution of 0.5° for the wheel angular position measurement.
The microcontroller board is equipped with (see figures 4.3, 4.4):

a. I^2C, Serial and Analog Digital ports for sensors purpose;

b. two ports for the motors control and encoder readings;

c. one barrel jack port for 9-12 V power supply;

d. RGB LEDs for status signaling;

e. interface for Raspberry Pi header;

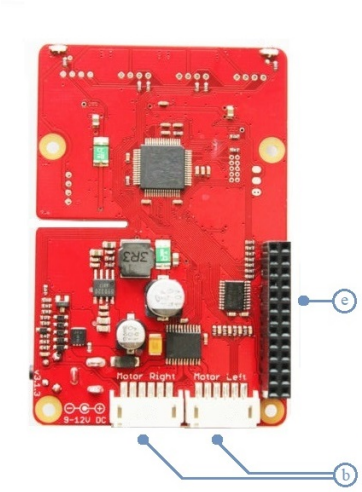


Figure 4.3: GoPiGo3 board: bottom side

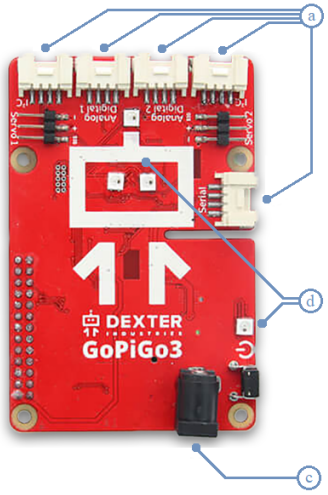


Figure 4.4: GoPiGo3 board: top side

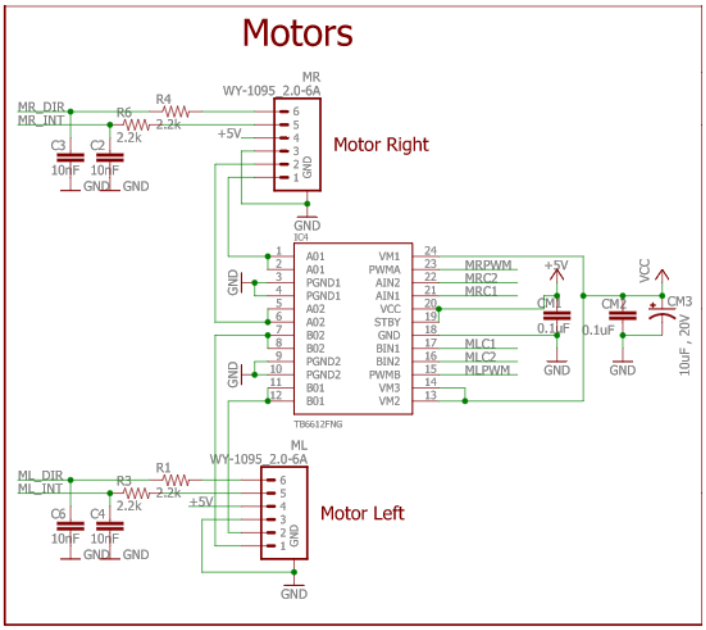


Figure 4.5: GoPiGo3 H-bridge integrated circuit scheme

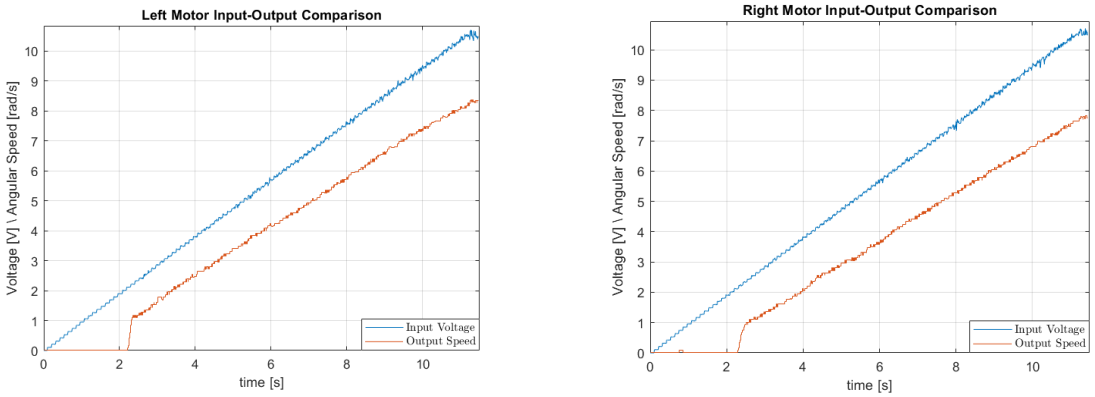
in addition it provides the possibility to mount several sensors which are not natives for GoPiGo3 applications, thanks to the General Purpose I/O (GPIO) pins.

Very important to the end of this experimental application is the integrated circuit that allows to actuate the motors, whose scheme is represented in figure 4.5. This integrated circuit implements two h-bridges (one for each motor), which are electronic circuits that allow to modulate the input voltage on the motor by means of a square wave with a variable duty cycle (DC), called PWM. More in detail, the scheme shows the two motor ports with 6 pins each: two of them are employed for the encoder infos transport, which are direction and count pulse, while the third one is used for $5\,V$ power supply, followed by one pin for ground and two for input voltage to the motor.
On the right side of the scheme the pins for input signals are represented: some of them are for the power supply, such as ones called *VMx*, used to receive the voltage that must be modulated and provided to the motors, and *VCC*, used for the chip power supply; other pins receive logical signals both for the voltage modulation (PWM) and for the motor rotational direction.
Some information about the PWM signal can be taken from the source [13], in which the square wave is displayed by means of an oscilloscope, showing that the

period of the PWM is $20\,ms$. In addition it is highlighted the functionality of the h-bridge, demonstrating that the voltage given as input to the motor changes from 0 to $|V_{max}|$ Volts by varying the square wave DC from $0\,\%$ to $100\,\%$.
Moreover a Raspberry Pi model 3B is used, it is equipped with an ARM Cortex-A7, a 32-bit microprocessor, and with a network card that supports WiFi connection by means of 802.11n protocol. The connectivity with the Raspberry is guaranteed also by an HDMI port and an Ethernet port.
A fundamental role in the robot control is played by sensors and actuators, hence their study is provided in the following sections. In this case both the measurement and the actuation are performed by means of high level programming interface (as will seen in section 4.2.1), which means that there is not a direct interaction with the hardware components. Nevertheless both hardware and software constraints must be understood in order to perform a proper reconstruction of Extended Plant behavior.

4.1.1 Actuators Study

The modulation of the motor input voltages requires the management of a PWM, the main consequence is that PWM duty cycle can't be varied within time intervals smaller than its period, which is $20\,ms$. In addition the actuator is affected by a resolution error since the PWM duty cycle is set with a maximum precision of one percentage point. Moreover, a dead-zone has been experienced for $|V_a| < 2.3\,V$



(a) left motor **(b)** right motor

Figure 4.6: wheels angular speed (red) according to motor input voltage (blue)

(this range can vary a lot), in fact input voltages within this range produce a zero response in wheel speed. What have just been said can be seen in figure 4.6, where the measured input voltage and wheel rotational speed are displayed, showing the output response to an input voltage ranging from 0 to its maximum value that

corresponds to the battery voltage supply: $V_{max} \approx 10.5\,V$. However the input shown in the figure is obtained by increasing PWM duty cycle of one percentage point per step, showing the resolution error problem which can be prized only for low values of the input voltage, where its measurements are cleaner than the highest values, in fact, as the duty cycle of the PWM increases, the noise is amplified and it becomes more difficult to see a clear jump between two steps.

4.1.2 Sensors Study

For measurement errors it is essential to remember that the encoders mounted on the motors are incremental with 6 pulses per revolution and, with a gear ratio $\tau = 120$, the wheel position angle measurement resolution is $\tilde{\alpha} = 0.5°$. At software level, a function which returns the pulse counter must be executed (see the Listing 4.3) and, letting $p(k)$ be the pulse count of the k-th sample, then the transition to the wheel rotation angle can be performed as follows:

$$\theta(k) = \frac{60°}{120} p(k) = p(k)\, 0.5°.$$

The speed measurements can be obtained by numerically differentiate the position samples, for this reason it will be referenced to as the derived measurement of the speed. Starting from the pulse counts of actual and previous wheel position, i.e. $p(k)$ and $p(k-1)$ respectively, in order to perform the backward Euler differentiation the following formula must be applied:

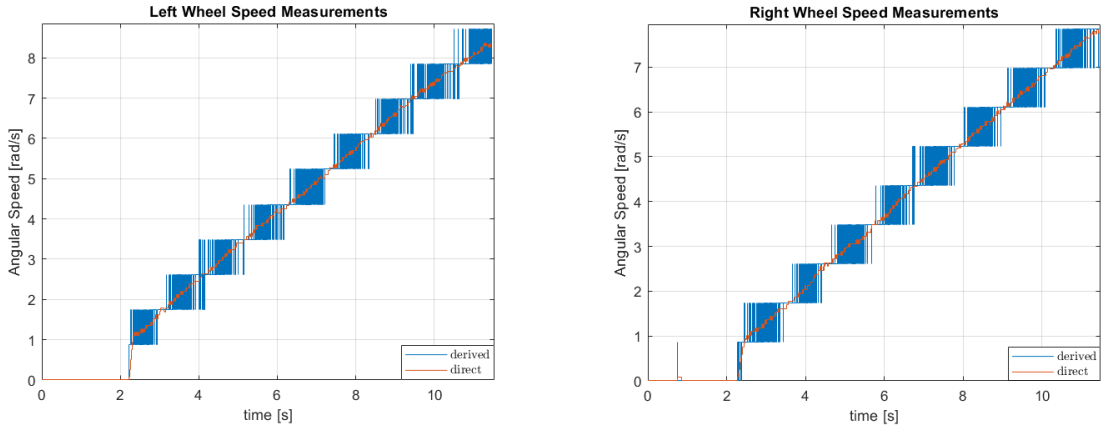
$$\omega(k) = \frac{(p(k) - p(k-1))\, 0.5°}{T_s}$$

where T_s is the sampling time. Please note that the resolution of this measurement is strongly dependent on the choice of sampling time, in fact the minimum distance between two consecutive pulse counts is 1, thus the parameter T_s is the only one which can determine how fine the resulting sample is.
Nevertheless another kind of speed measurement is obtained by calling a function which directly returns the pulse frequency. In the second case, letting $f(k)$ be the pulse frequency read at k-th step, then the wheel speed can be computed as

$$\omega(k) = \frac{f(k)}{2}$$

where 2 are the number of pulses needed for one wheel degree. In addition, since the pulse frequency f is represented on a 16 bits integer variable (see the Listing 4.3), it is simple to understand that the resolution error is $e_r = 0.5\, deg/s$ for this measurement method.

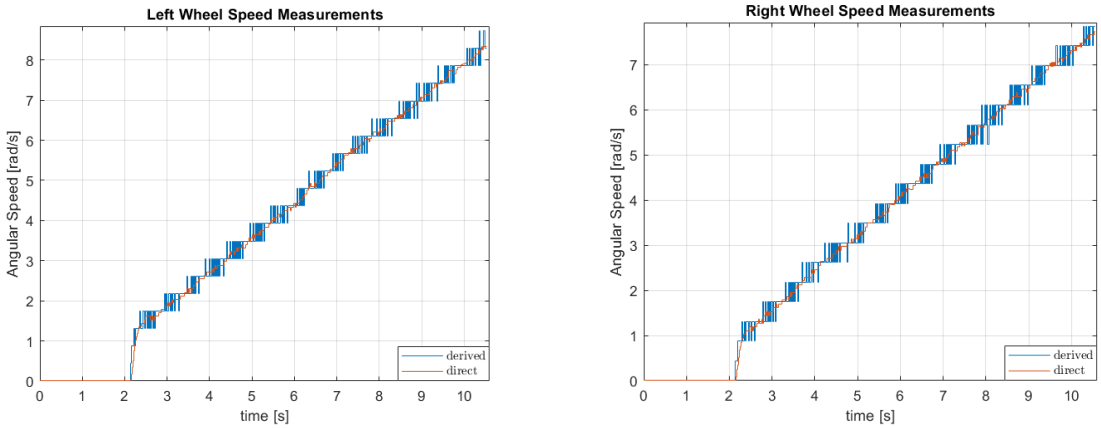
The two measurement methods are compared below, both are performed simultaneously on the same data acquisition process obtained by providing to the motors an input voltage ranging from 0 up to V_{max} as in the example above.



(a) left wheel **(b)** right wheel

Figure 4.7: wheels direct (red) and derived (blue) angular speed measurements with $T_s = 10\,ms$.

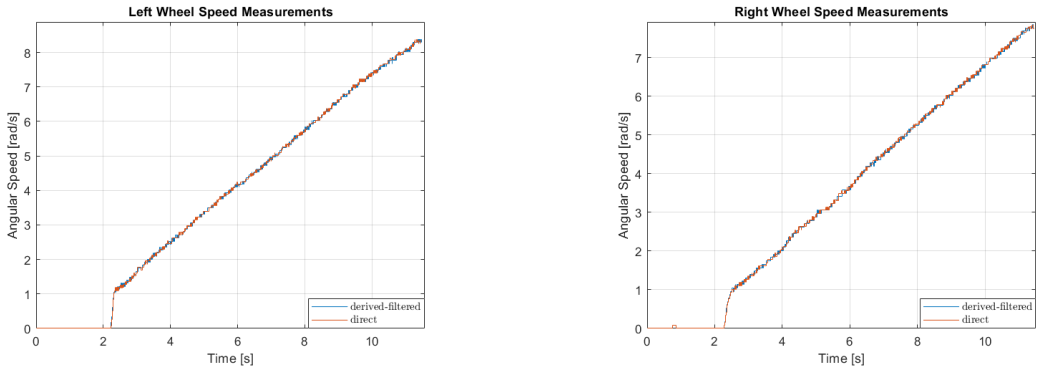
Figure 4.7 shows the speed measurements for the left and right wheels obtained at a sampling time $T_s = 10\,ms$. It is clear that, in this case, the direct measurements have a better resolution than the derived ones but, on the other hand, the direct measurements are subject to a slight delay with respect to the other one that is experienced especially in the rising edge around the time instant $t = 2\,s$. Figure 4.8 shows the results of a measurement process identical to that displayed in the previous figure, with the difference that a sampling time of $20\,ms$ is used for this case. Both the plots show a substantial improvement in the resolution of the measurements, confirming what was said previously. Anyway, in this case too, the direct measurement behaves better than the derived one, except for the delay in the rising edge, which is even more pronounced than in the previous case.
In conclusion, although the derived measurement has a greater responsiveness with respect to the direct measurement, it still has a too low resolution, especially for the sampling time $T_s = 10\,ms$, which is the one chosen to implement the remote control system; for this reason, the direct measurement method is the one used for this project.
Before going on, a last comment is needed about the two kinds of measurements, so that the choices made in the following sections can be well understood. In figure 4.9 you can see the same comparison presented in figure 4.7 with the difference that the derived measurements are filtered by means of the *Moving Average* technique



(a) left wheel **(b)** right wheel

Figure 4.8: wheels direct (red) and derived (blue) angular speed measurements with $T_s = 20\,ms$.

implemented with a sliding window of 10 slots. As can be prized from the figure, by applying this kind of filter to the derived measurements, the blue line and the red line become very similar both for the trend of the samples and for their resolution. It is very important to underline this aspect because it provides a solution for a software emulation of the sensors behavior.



(a) left wheel **(b)** right wheel

Figure 4.9: wheels direct (red) and filtered derived (blue) angular speed measurements with $T_s = 10\,ms$.

4.2 Software Specifications

The operating system installed in the microSD plugged in the Raspberry board is called DexterOS and is a Raspbian-like OS owned by Dexter Industires.
DexterOS allows to use the Raspberry as a server, indeed by digiting its IP address, which can also be obtained by resolving the hostname "*dex.local*", on the search bar of the browser it will answer with a web page which lets to access remotely to the robot in two different modes:

- Desktop view, on TCP port 8001, which offers a classical GUI-based interface, very convenient for developing code through graphical IDEs;

- Terminal view, on TCP port 4200, which allows interactions by means of shell commands only, providing a lighter and more energy-saving solution than the first one.

In addition to the OS, some developers tools are included in the installation package, ranging from ad-hoc graphical programming languages for beginners to more complex languages such as `C++`, `C#`, Python or Java, providing libraries which allow to communicate with motors, encoders and other sensors eventually attached to board ports. In particular, the `C++` libraries will be analyzed since it is the language chosen for the development of the project described in the next chapters.

4.2.1 GoPiGo3 `C++` libraries

Some libraries dedicated to robot control are made available for code development, whose symbols are declared in file "*GoPiGo3.h*" and are defined in "*GoPiGo3.cpp*" one.
The presence of an operating system simplifies a lot the management of an I/O device, in particular the Linux-based ones use pseudo files, typically placed in directory "*/dev*" , allowing to treat the communication with a device as a read/write procedure on a file, [14]; the piece of code in 4.1 confirm what said, showing how the transfer of the data can be handled. The just mentioned code is directly taken from file "*GoPiGo3.h*" and it is possible to see that the I/O device is treated as a file since the *'`#define`'* directive saves into '`SPIDEV_FILE_NAME`' the path of a pseudo file which represents the SPI (Serial Peripheral Interface) device. Finally, as for a normal file, it is opened through the system call *`open()`* which returns a handle used as a file identifier from that moment on. Very important is the structure *'`spi_xfer_struct`'*, which is used as a buffer both for transmit and for receive data, providing the length of the transferred data and optional infos related to the communication , such as the time delay or the bit rate, [15].

Listing 4.1: GoPiGo3.h: SPI device opening

```
1
2     ...
19    #define SPI_TARGET_SPEED 500000 // SPI target speed of 500kbps
20
21    #define SPIDEV_FILE_NAME "/dev/spidev0.1" // File name of SPI
22
23      ...
46    struct spi_ioc_transfer spi_xfer_struct; // SPI transfer struct
47    uint8_t spi_array_out[LONGEST_SPI_TRANSFER]; // SPI out array
48    uint8_t spi_array_in[LONGEST_SPI_TRANSFER];  // SPI in array
49    // Set up SPI. Open the file, and define the configuration.
50    int spi_setup(){
51       spi_file_handle = open(SPIDEV_FILE_NAME, O_RDWR);
52
53       if (spi_file_handle < 0){
54          return ERROR_SPI_FILE;
55       }
56
57       spi_xfer_struct.cs_change = 0; //Keep CS activated
58       spi_xfer_struct.delay_usecs = 0; //delay in us
59       spi_xfer_struct.speed_hz = SPI_TARGET_SPEED; //Speed
60       spi_xfer_struct.bits_per_word = 8; // bits per word
61
62       return ERROR_NONE;
63    }
64
65      ...
```

The function *'spi_transfer_array()'* plays a central role in communicating with I/O devices, since allows to send any kind of data collected in the user memory location pointed by the parameter *'outArray'* and to receive data saving it on the user memory location pointed by the parameter *'inArray'*. The piece of code in 4.2 highlights what just said, showing that the buffers pointers are saved in the structure mentioned above which, in turn, is passed to the system call *ioctl()*, in charge of performing the communication with the driver.

Listing 4.2: GoPiGo3.h: SPI transfer array function

```
1     ...
65    // Transfer length number of bytes. Write from outArray, read to inArray.
66    int spi_transfer_array(uint8_t length, uint8_t *outArray, uint8_t *inArray){
67    spi_xfer_struct.len = length;
68    spi_xfer_struct.tx_buf = (unsigned long)outArray;
69    spi_xfer_struct.rx_buf = (unsigned long)inArray;
70
71       if (ioctl(spi_file_handle, SPI_IOC_MESSAGE(1), &spi_xfer_struct) < 0) {
72          return ERROR_SPI_FILE;
73       }
74
75    return ERROR_NONE;
76    }
77
78      ...
```

In simple terms, the function exposed in the code 4.2 is the backbone of all the C++ GoPiGo3 library, since every time it is needed to transmit data with HW devices (i.e. motors commands, battery information or sensors readings) this function must be called. However the code exposed above is at a very low level of this library and it is not needed to directly deal with it, nevertheless it is essential to know it in order to understand the functioning of higher level code. The library provides a class named GoPiGo3 that doesn't contain member variables of interest but exposes a series of setter and getter functions which have the role of, respectively, sending to and receiving from peripherals. For initialization of the class it is used the function "*detect()*", needed to make sure the robot is connected and the firmware is up to date, on the other hand the function "*reset_all()*" must be called to reset all the peripherals, included motors and encoders.

Listing 4.3: GoPiGo3.h: some member functions of GoPiGo3 class

```
1       ...
265     int get_voltage_battery(float &voltage);
266
267       ...
270     // Set the motor PWM power
271     int set_motor_power(uint8_t port, int8_t power);
272
273       ...
270     // Get the motor status. State, PWM power, encoder position, and speed (in
         degrees per second)
271     int get_motor_status(uint8_t port, uint8_t &state, int8_t &power,
                             int32_t &position, int16_t &dps);
272     // Offset the encoder position. By setting the offset to the current position,
         it effectively resets the encoder value.
273     int offset_motor_encoder(uint8_t port, int32_t position);
274
275     // Get the encoder position
276     // Pass the port and pass-by-reference variable where the encoder value will be
          stored. Returns the error code.
277     int get_motor_encoder(uint8_t port, int32_t &value);
278
279       ...
```

In code listed in 4.3 the member functions of class `GoPiGo3`, used for this project, are presented:

- *get_voltage_battery()* is used to get the voltage which must be modulated by the PWM signal, in other words it represents the maximum supply voltage for the motors;

- *set_motor_power()* is the method used to set the PWM duty cycle for a motor by means of the parameter *power*. It is worth to notice that it is a signed 8 bits variable, this means that (i) when it assumes negative values the motor rotates in the opposite verse and (ii) all its values higher than 100

in magnitude, are considered to be equal to 100 since the magnitude of this parameter represents the percentage of the PWM duty cycle;

- *get_motor_status()* is used to obtain some measurements related to a specified motor, in particular it is needed for angular position and angular speed measurements saved in variables ***position*** and ***dps***, passed as rvalue reference to the function. In first parameter is saved the position of the encoder in terms of pulses count, in second one the degrees per second. Please note that this is the function used to obtain the direct measurement discussed in section 4.1.2;

- *offset_motor_encoder()* is to set the encoder position offset;

- *get_motor_encoder()* is used to measure the encoder position. It is used for the derived speed measurement.

Finally in the last three methods it is possible to note the same parameter ***port*** which is in charge to define the motor where must be performed the function operation; thus it can assume only two values represented by the constants defined in the file *GoPiGo3.h* which are `MOTOR_LEFT` and `MOTOR_RIGHT`.

Chapter 5

Model and Control of Motor-Wheel System

At this point the practical application of what has been said about the EMC theory must be addressed. Particularly, this chapter will deal with a dynamic analysis of the real motor-wheel system in order to provide a model for the EMC.

5.1 Motor-Wheel Model Definition

Since no detailed information has been provided regarding the DC motor used for the robot in question, a generic model is considered so that appropriate state space equations can be derived. The figure 5.1 represents a scheme of a DC motor which is attached to a load, in this case the wheel, by means of a gearbox with ratio τ, forming what in this thesis will be referred to as *motor-wheel* system. More in detail, the model considered does not take into account any type of elasticity, i.e. all the components are assumed to be stiff, and any type of slack in the gearbox is neglected. Finally, it should be noted that some of the disturbing torques acting on the wheel, such as, for example, the rolling friction, are grouped within the defined torque T_l; on the other hand, with T_r the disturbing torques that would act directly on the motor side are identified. At this point let's analyze the dynamic equations related to this model and used for the construction of the EM. In order to obtain the final equations, let r_m and r_w be the radii of the motor-side and wheel-side gears respectively, and let f be the force of interaction between the two gears; then

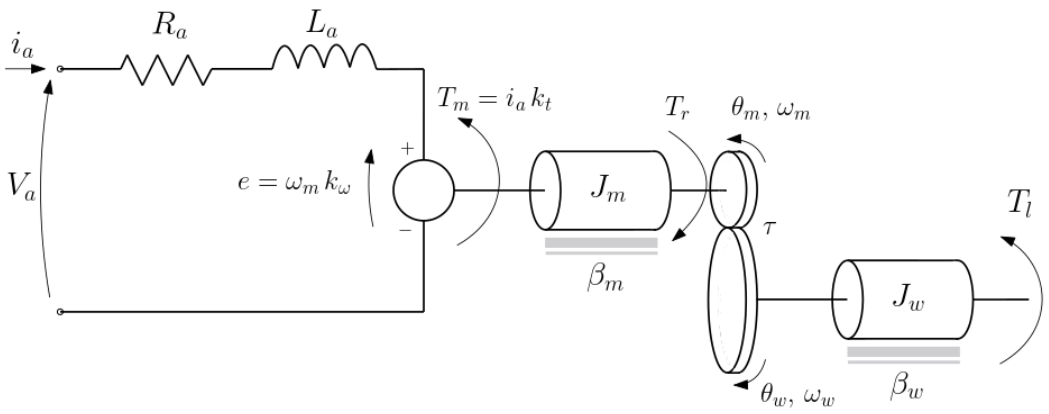


Figure 5.1: scheme of DC motor and wheel system

the following equations are provided:

$$\begin{cases} V_a(t) = R_a\, i_a(t) + L_a\, \dot{i}_a(t) + k_\omega \omega_m(t) \\ J_m\, \dot{\omega}_m(t) = k_t\, i_a(t) - \beta_m \omega_m(t) - T_r(t) - r_m\, f(t) \\ J_w\, \dot{\omega}_w(t) = r_w\, f(t) - \beta_w \omega_w(t) - T_l(t) \\ \dot{\theta}_m(t) = \omega_m(t) \end{cases} \tag{5.1}$$

where the first equation is obtained by applying the voltage equilibrium equation at the electrical part, while the following ones are obtained by the analysis of the mechanical part. Performing suitable substitutions and rearrangements on second and third equations, and reminding that $\tau = r_w/r_m = \omega_m(t)/\omega_w(t)$, it can be obtained:

$$\underbrace{\left(J_m + \frac{J_w}{\tau^2}\right)}_{=J_{eq}} \dot{\omega}_m(t) = k_t\, i_a(t) - \underbrace{\left(\beta_m + \frac{\beta_w}{\tau^2}\right)}_{=\beta_{eq}} \omega_m(t) - \underbrace{\left(T_r(t) + \frac{T_l(t)}{\tau}\right)}_{=T_d(t)} . \tag{5.2}$$

This last equation speaks volumes about the mechanical action of the gearbox, in fact it can be noticed that the toques coming from the wheel-side, which react to the motor action, are reduced by factors τ^2 and τ; it means that a relatively high reduction factor (as in this case, where $\tau = 120$) will cause a mechanical damping of the reactions and disturbances coming from the wheel side.
Anyway, at this point the final state-space equations can be obtained by using the

5.2, which lets to write:

$$\begin{cases} \dot{i_a}(t) = -\dfrac{R_a}{L_a} i_a(t) - \dfrac{k_\omega}{L_a} \omega_m(t) + \dfrac{1}{L_a} V_a(t) \\ \dot{\omega}_m(t) = \dfrac{k_t}{J_{eq}} i_a(t) - \dfrac{\beta_{eq}}{J_{eq}} \omega_m(t) - \dfrac{1}{J_{eq}} T_d(t) \\ \dot{\theta}_m(t) = \omega_m(t) \end{cases} \tag{5.3}$$

Finally, passing to the *Laplace* domain, the resulting equations allow to sketch the block scheme in figure 5.2 .

$$\begin{cases} i_a(s) = \dfrac{1}{R_a + sL_a} \left(V_a(s) - k_\omega \, \omega_m(s) \right) \\ \omega_m(s) = \dfrac{1}{\beta_{eq} + sJ_{eq}} \left(k_t \, i_a(s) - T_d(s) \right) \\ \theta_m(s) = \dfrac{1}{s} \omega_m(s) \end{cases} \tag{5.4}$$

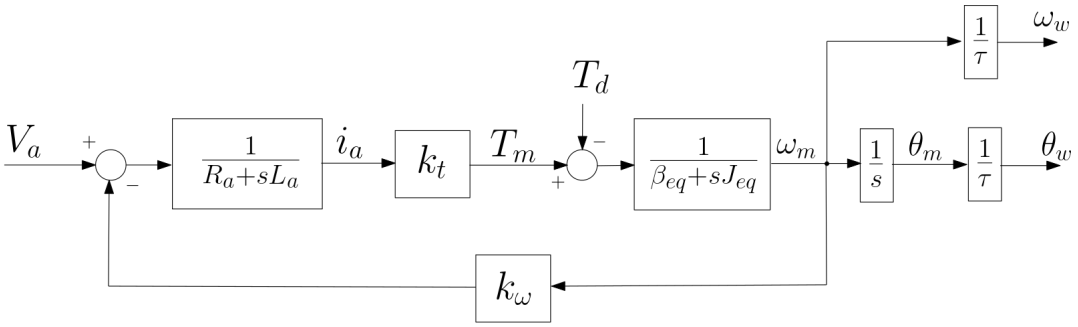


Figure 5.2: block scheme of DC motor and wheel system

5.1.1 Model Evaluation

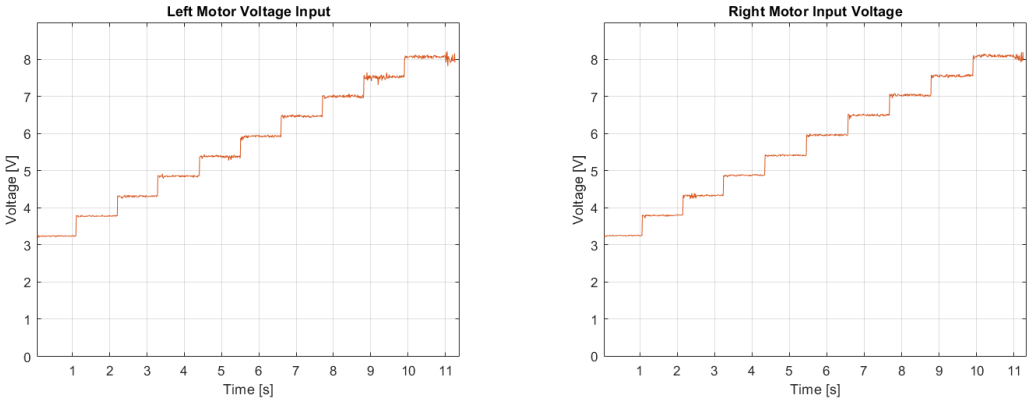
At this point it can be proper to carry out an evaluation of the model taken into account. To this aim the values of the parameters are needed but no official datasheets are provided by GoPiGo3 motors suppliers, therefore a parameter identification should be performed. In this regard, the work carried out in the Source [13] can be very useful since a detailed process for estimating the parameters of GoPiGo3's motors is provided in there. It is very important to specify that the purpose here is not to use parameters which perfectly fit the motors behavior, it would be impossible since from one motor to another there could be structural differences that could cause considerable variations in their parameters. Nevertheless, it was

decided to use the parameters estimated in the above mentioned thesis, since in the past chapters it has been widely shown how EMC is able to compensate for parametric errors.
The parameters used for this thesis are exposed in the table below. In order to

Parameter	Unit of Measure	Value
β_{eq}	$Nm\,(s/rad)$	4.3116×10^{-7}
J_{eq}	$kg\,m^2$	4.6743×10^{-7}
L_a	H	0.2152
R_a	Ω	18.573
k_ω	Vs	0.0134
k_t	$Nm\,A^{-1}$	0.0134

Table 5.1: DC Motor Parameters

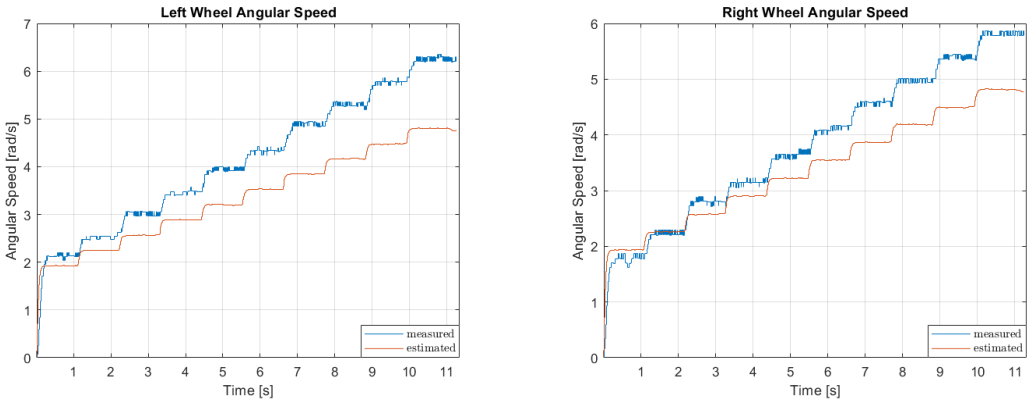
evaluate the resulting model, the measurements of rotational speed the wheels are compared to the response obtained by the model implemented in SIMULINK, to which the same input given to the motors is provided. The SIMULINK project is based on the block scheme of figure 5.2, the results are displayed below.



(a) left motor **(b)** right motor

Figure 5.3: input voltage obtained by ranging PWM duty cycle from 30% to 75% with increment of 5% each jump.

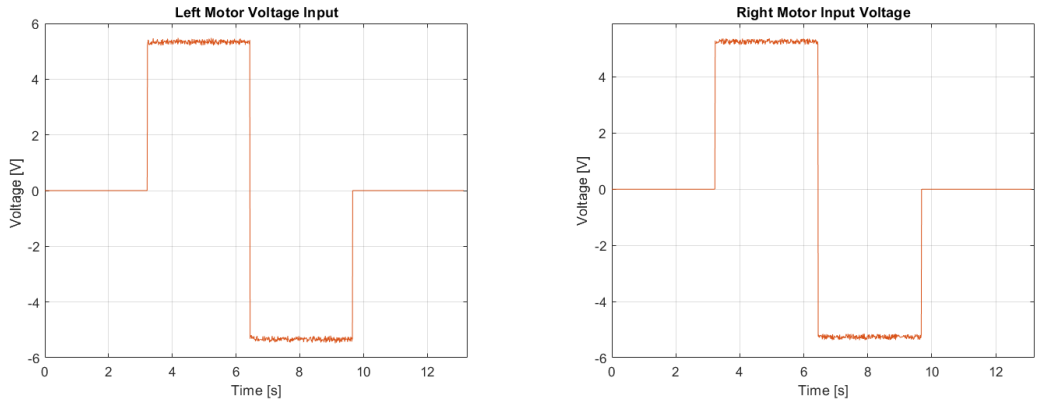
Figures 5.3 and 5.4 display the the input voltage provided to the motors and the resulting angular speeds. As can be seen, the input is obtained by increasing the PWM duty cycle of 5 percentage points for 9 times starting from 30%. The first



(a) left wheel **(b)** right wheel

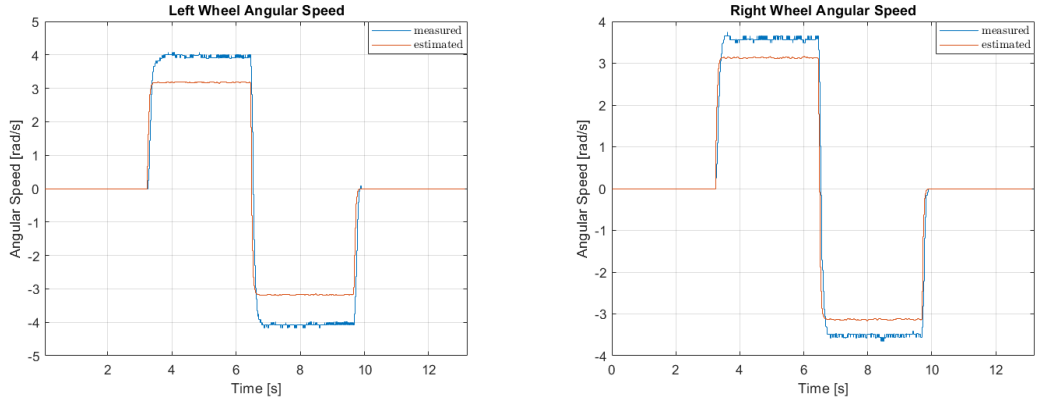
Figure 5.4: comparison between measured wheels angular speed output (blue) and model output (red) for stairs input.

aspect that must be highlighted is the different response of the two motors, in fact, as can be seen in figures 5.4a and 5.4b, the "stairs" represented by the two blue lines have two different slopes, confirming that there are differences between parameters of the two motors. Moreover as these two figures point out, there is a substantial difference between model response and measurements for both the right and left wheels, suggesting a gain error caused by a bad choice of parameters. Anyway, as said above, the EMC control methodology is able to reject parameter errors, thus, as we will see, the parameters chosen above are more than sufficient. The aspect of the model that needs to be evaluated more carefully is its responsiveness, whose evaluation can be performed with the next example. Figure 5.5 represents another voltage input provided to the motors, obtained by setting the PWM DC at the values {0, 50, -50, 0}, in this order. Also in this case the figure 5.6 points out a gain error already highlighted in the previous case, but the goal here is to show the behavior of the model transient with respect to the measurements. About this, the estimated output seems to be slightly in advance wrt the measurements, but please remember that the direct measurement method provides a delay in the system output (see 4.1.2). In any case, the transient of the estimated output is very similar to that given by the measurements, therefore, although the chosen parameters don't give a precise estimate of the output, the model can be considered to be acceptable for its application on the EM.



(a) left motor **(b)** right motor

Figure 5.5: input voltage obtained by PWM duty cycle values sequence: {0, 50, -50, 0}%.



(a) left wheel **(b)** right wheel

Figure 5.6: comparison between measured wheels angular speed output (blue) and model output (red) for double port input.

5.2 Asynchronous-EMC Design for Motor-Wheel System

This section deals with the design choices taken for the construction of the Asynchronous-EMC which is able to control the wheel-motor system described above in asynchronous measurements and commands scenario. It is worth to notice that the Asynchronous-EMC can be considered as a generalization of the canonical EMC, therefore their designs are equivalent and, in order to use the

Asynchronous-EMC design for a normal EMC application, it suffices to provide a constant sampling time T. It is important to specify this aspect because, after the design phase, there will be an experimentation phase carried out on software simulations first (both for asynchronous and synchronous systems) and then directly on the robot with an embedded application. Once these two test phases have been passed, we move on to experimenting with remote control, by testing the design of the control unit with the variable sampling time in a real NCS.

5.2.1 Embedded Model Design

Let's start from the design of the EM, which includes both *Actual* and *Reference Dynamics*. The model used here was widely discussed in section 5.1, whose state-space equations defined in 5.3 are employed in order to obtain the discrete time model. In addition to those equations both driving noise and disturbance states must be added in order to obtain the complete *Actual Dynamics* definition. In particular, for this design, a single disturbance state is accounted, and it is modelled as an acceleration disturbance. Therefore, the state $\mathbf{x}$ has dimension $n = 3$ and, in order to design a *dynamic* noise estimator, the driving noise is chosen to have dimension $n_w = 2$, with the first component directly added to the armature current differential equation while the second one is added to the disturbance state differential equation. Reminding that $\omega_m = \tau\,\omega_w$, the equations below represent what has just been said.

$$\begin{cases} \dot{i_a}(t) = -\dfrac{R_a}{L_a} i_a(t) \; - \dfrac{\tau\, k_\omega}{L_a} \omega_w(t) + \dfrac{1}{L_a} V_a(t) + w_1(t) \\ \dot{\omega}_w(t) = \dfrac{k_t}{\tau\, J_{eq}} i_a(t) - \dfrac{\beta_{eq}}{J_{eq}} \omega_w(t) + x_d(t) \\ \dot{x}_d(t) = w_2(t) \end{cases} \tag{5.5}$$

First of all, please notice that the position is not included in the state-space in order to have a lower complexity, also note that the disturbance torque T_d can be considered to be included into the more generic acceleration disturbance x_d. At this point, applying the forward Euler discretization method to the just displayed equations, the DT equations can be written as follows.

$$\begin{cases} i_a(t_i + T_i) = \left(1 - T_i \dfrac{R_a}{L_a}\right) i_a(t_i) \; - T_i \dfrac{\tau\, k_\omega}{L_a} \omega_w(t_i) + \dfrac{T_i}{L_a} V_a(t_i) + T_i\, w_1(t_i) \\ \omega_w(t_i + T_i) = \left(1 - T_i \dfrac{\beta_{eq}}{J_{eq}}\right) \omega_w(t_i) + T_i \dfrac{k_t}{\tau\, J_{eq}} i_a(t_i) + T_i\, x_d(t_i) \\ x_d(t_i + T_i) = x_d(t_i) + T_i\, w_2(t_i) \end{cases} \tag{5.6}$$

As can be seen, the controllable state is composed by armature current i_a and wheel rotational speed ω_w, letting V_a be the system input and ω_w be its output, the *Controllable Dynamics* can be written with the matrix form below:

$$\begin{aligned}
\mathbf{x}_c(t_i + T_i) &= \overbrace{\begin{bmatrix} 1 - T_i \dfrac{R_a}{L_a} & -T_i \dfrac{\tau\, k_\omega}{L_a} \\ T_i \dfrac{k_t}{\tau\, J_{eq}} & 1 - T_i \dfrac{\beta_{eq}}{J_{eq}} \end{bmatrix}}^{A_{ci}} \mathbf{x}_c(t_i) + \overbrace{\begin{bmatrix} \dfrac{T_i}{L_a} \\ 0 \end{bmatrix}}^{B_{ci}} u(t_i) + \mathbf{d}(t_i) \\
y_m(t_i) &= \underbrace{\begin{bmatrix} 0 & 1 \end{bmatrix}}_{C_c} \mathbf{x}_c(t_i), \qquad \begin{bmatrix} i_a \\ \omega_w \end{bmatrix}(0) = \mathbf{x}_{c0}
\end{aligned} \tag{5.7}$$

where $\mathbf{d}$, as discussed in the theoretical part, represents the modelled disturbances acting on the system, including both the driving noise $\mathbf{w}$ and the disturbance state x_d; it can be obtained by the *Disturbance Dynamics* written in the following matrix form:

$$\begin{aligned}
x_d(t_i + T_i) &= x_d(t_i) + \overbrace{\begin{bmatrix} 0 & T_i \end{bmatrix}}^{G_{di}} \mathbf{w}(t_i), \qquad x_d(0) = 0 \\
\mathbf{d}(t_i) &= \underbrace{\begin{bmatrix} 0 \\ T_i \end{bmatrix}}_{H_{ci}} x_d(t_i) + \underbrace{\begin{bmatrix} T_i & 0 \\ 0 & 0 \end{bmatrix}}_{G_{ci}} \mathbf{w}(t_i)
\end{aligned} \tag{5.8}$$

from which results $A_{di} = 1, \forall\, i$. Finally, all the matrices needed for the construction of the *Actual Dynamics* are found, thus all the matrices A_i, B_i, G_i can be constructed as shown in 2.32.
At this point it is very simple to build the *Reference Dynamics*, in fact it suffices to copy the Controllable Dynamics by neglecting the disturbances, obtaining the matrix form equations written below.

$$\begin{aligned}
\underline{\mathbf{x}}(t_i + T_i) &= \begin{bmatrix} 1 - T_i \dfrac{R_a}{L_a} & -T_i \dfrac{\tau\, k_\omega}{L_a} \\ T_i \dfrac{k_t}{\tau\, J_{eq}} & 1 - T_i \dfrac{\beta_{eq}}{J_{eq}} \end{bmatrix} \underline{\mathbf{x}}(t_i) + \begin{bmatrix} \dfrac{T_i}{L_a} \\ 0 \end{bmatrix} \underline{u}(t_i) \\
\underline{y}(t_i) &= \begin{bmatrix} 0 & 1 \end{bmatrix} \underline{\mathbf{x}}(t_i), \qquad \begin{bmatrix} \underline{i}_a \\ \underline{\omega}_w \end{bmatrix}(0) = \underline{\mathbf{x}}_0
\end{aligned} \tag{5.9}$$

5.2.2 Reference Generator Design

Let's remind that the aim of the *Reference Generator* is to generate the input $\underline{u}$ such that the output $\underline{y}$ follows the reference r with a "legal" trajectory, where the term "legal" implies that the constraints imposed on both the states and the input must be respected. In this specific case an input constraint is imposed by the actuator, that is its saturation to the value $|V_{max}| = 10.5\,V$ corresponding to the battery voltage. Anyway a control of the reference dynamics output must be performed and an usual static state-feedback control law can be designed; the resulting Reference Generator is sketched in the figure 5.7. To this aim the matrix

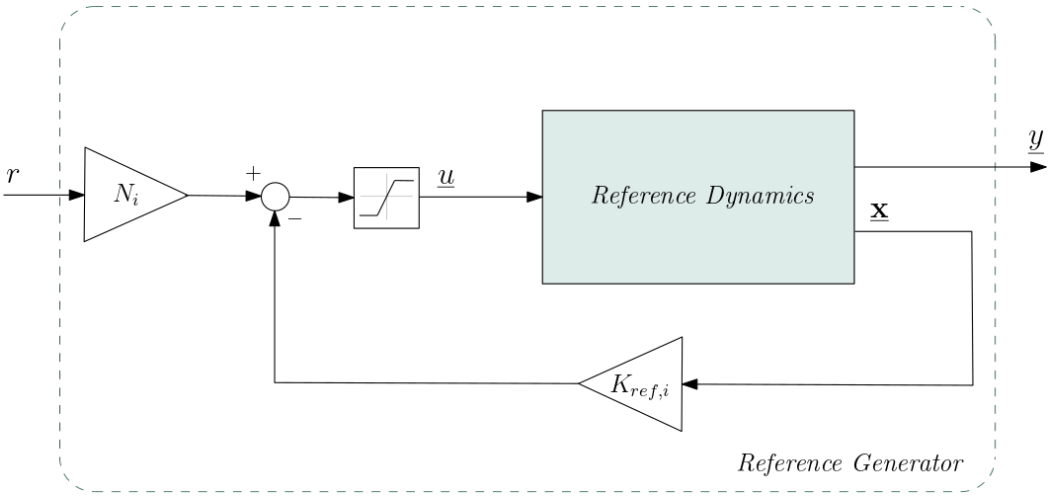


Figure 5.7: Reference Generator with saturation of input $\underline{u}$ and state-feedback control.

$K_{ref,i} = [k_{i1}\ k_{i2}]$ must be computed using pole placement technique, which provides the following solution:

$$k_{i1} = \frac{1}{b_{ci1}}(a_{ci11} + a_{ci22} - \lambda_{ref,i1} - \lambda_{ref,i2})$$

$$k_{i2} = \frac{1}{a_{ci21}b_{ci1}}(\lambda_{ref,i1}\lambda_{ref,i2} - a_{ci11}a_{ci22} + a_{ci12}a_{ci21} + a_{ci22}b_{ci1}k_{i1})$$

where $\lambda_{ref,i1}$ and $\lambda_{ref,i2}$ are the DT domain eigenvalues to place at instant t_i. Therefore the matrix N_i is needed in order to obtain a unitary dc-gain in the transfer function between r and $\underline{y}$, for this reason N_i is computed as follows:

$$N_i = \left[C_{ci}\left[\mathbf{I} - (A_{ci} - B_{ci}\,K_{ref,i})\right]^{-1} B_{ci}\right]^{-1}.$$

In the figure the saturation block can be recognised, and it is placed there in order to guarantee that the reference input is within the limits imposed by the maximum

battery voltage supply, namely $-10.5 \leq \underline{u} \leq 10.5$.

5.2.3 Noise Estimator Design

As already mentioned, we want to implement a dynamic noise estimator, this implies that a model error filtering is performed by an integrator included into the driving noise estimator. Let's remind that, for this kind of noise estimator, the observer closed-loop states matrix A_m is constructed as follows:

$$A_{mi} = \begin{bmatrix} A_i - G_i\, L_i\, C_i & G_i\, M_{mi} \\ -C_i & 1 - \beta_i \end{bmatrix}$$

where $M_{mi} = \begin{bmatrix} m_{i0} \\ m_{i1} \end{bmatrix}$, $L = \begin{bmatrix} l_{i0} \\ l_{i1} \end{bmatrix}$ and β are the unknowns of the pole placement problem. Since the dimension of $\mathbf{x}$ is $n = 3$, the number of eigenvalues to place are $dim(A_{mi}) = 4$ but there are 5 unknowns, therefore at least one parameter must be set to an arbitrary value. In detail, by choosing $l_{i0} = 0$, $\forall\, i$, it is possible to obtain the remaining parameters in function of the desired eigenvalues in the following way:

$$\begin{aligned}
\beta_i &= a_{i11} + a_{i22} + a_{i33} - \alpha_{i3} + 1 \\
l_{i1} &= -\frac{1}{a_{i23} g_{i32}}[(1-\beta_i)(a_{i11} + a_{i22} + a_{i33}) + a_{i11}a_{i22} + a_{i11}a_{i33} + a_{i22}a_{i33} - \\
&\qquad - a_{i12}a_{i21} - \alpha_{i2}] \\
m_{i1} &= \frac{1}{a_{i23} g_{i32}(a_{i33} - a_{i11})}[(1-\beta_i)(a_{i11}a_{i33}^2 + a_{i22}a_{i33}^2 + (a_{i33} - a_{i11})a_{i23}g_{i32}l_{i1}) + \\
&\qquad + \alpha_{i0} - a_{i33}(\alpha_{i1} - a_{i11}a_{i22}a_{i33} + a_{i12}a_{i21}a_{i33} - a_{i11}a_{i23}g_{i32}l_{i1})] \\
m_{i0} &= \frac{1}{a_{i21} g_{i11}}[(1-\beta_i)(a_{i11}a_{i22} + a_{i11}a_{i33} + a_{i22}a_{i33} - a_{i12}a_{i21} + a_{i23}g_{i32}l_{i1}) - \alpha_{i1} + \\
&\qquad + a_{i11}a_{i22}a_{i33} - a_{i12}a_{i21}a_{i33} + a_{i23}g_{i32}(a_{i11}l_{i1} - m_{i1})]
\end{aligned}$$

being $P_i(\lambda) = \lambda^4 + \alpha_{i3}\,\lambda^3 + \alpha_{i2}\,\lambda^2 + \alpha_{i1}\,\lambda + \alpha_{i0}$ the desired characteristic polynomial at time step t_i, with the coefficients depending on the desired DT eigenvalues as shown below.

$$\begin{aligned}
\alpha_{i0} &= \lambda_{mi1}\lambda_{mi2}\lambda_{mi3}\lambda_{mi4} \\
\alpha_{i1} &= \lambda_{mi1}\lambda_{mi2}\lambda_{mi3} + \lambda_{mi1}\lambda_{mi2}\lambda_{mi4} + \lambda_{mi1}\lambda_{mi3}\lambda_{mi4} + \lambda_{mi2}\lambda_{mi3}\lambda_{mi4} \\
\alpha_{i2} &= \lambda_{mi1}\lambda_{mi2} + \lambda_{mi1}\lambda_{mi3} + \lambda_{mi1}\lambda_{mi4} + \lambda_{mi2}\lambda_{mi3} + \lambda_{mi2}\lambda_{mi4} + \lambda_{mi3}\lambda_{mi4} \\
\alpha_{i3} &= \lambda_{mi1} + \lambda_{mi2} + \lambda_{mi3} + \lambda_{mi4}
\end{aligned}$$

At this point, once the unknowns are computed, the estimation of the driving noise can be performed by applying the following formulae

$$\begin{aligned} &\bar{w}_1(t_i) = m_{i0}\, q(t_i)\,, \\ &\bar{w}_2(t_i) = l_{i1}\, \bar{e}(t_i) + m_{i1}\, q(k)\,, \\ &q(t_i + T_i) = \bar{e}(t_i) + (1 - \beta_i) q(t_i), \quad q(0) = 0\,. \end{aligned} \tag{5.10}$$

5.2.4 Control Law Design

Let's remind that for the design of *Control Law* both the conditions 2.36 and 2.37, must be achieved. The first one can be accomplished by performing the usual pole placement which allows to compute the matrix $K_i = [k_{i1}\ k_{i2}]$ in function of the desired poles λ_{ci1} and λ_{ci2}. In particular the solution for this problem is exactly the same of the one obtained for the Reference Generator design, in fact they are the same problem.

$$\begin{aligned} k_{i1} &= \frac{1}{b_{ci1}}(a_{ci11} + a_{ci22} - \lambda_{ci1} - \lambda_{ci2}) \\ k_{i2} &= \frac{1}{a_{ci21} b_{ci1}}(\lambda_{ci1}\lambda_{ci2} - a_{ci11}a_{ci22} + a_{ci12}a_{ci21} + a_{ci22}b_{ci1}k_{i1}) \end{aligned}$$

Therefore the components of K_i are computed as shown above while the matrices $Q = \begin{bmatrix} q1 \\ q2 \end{bmatrix}$ and $M = m$ (in this case M is a scalar) must be computed by solving the following Devison-Francis condition

$$\begin{bmatrix} q_1 \\ T_i + q_2 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 - T_i \dfrac{R_a}{L_a} & -T_i \dfrac{\tau\, k_w}{L_a} & \dfrac{T_i}{L_a} \\ T_i \dfrac{k_t}{\tau\, J_{eq}} & 1 - T_i \dfrac{\beta_{eq}}{J_{eq}} & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ m \end{bmatrix}$$

which provides the following solution for the unknowns q_1, q_2, m:

$$\begin{aligned} q_1 = &\frac{\tau\, J_{eq}}{k_t}, \quad q_2 = 0\,, \\ &m = q_1\, R_a\,. \end{aligned}$$

Note that these parameters are independent from the sampling time T_i. At this point all is needed to build the Control Law is available, it suffices then to use the formulae 2.17 and 2.19 (properly adapted to the asynchronous case) such that the desired command u is provided to both the Plant and the Embedded Model.

Chapter 6

Wheels Rotational Speed Control Simulations

At this point it is possible to practice the asynchronous-EMC designed above, first of all starting from simulations, whose aim is to test the closed loop system tuning the eigenvalues of observer, controller and reference generator, and then moving on to the real application. Please note that although the final purpose is to implement a remote control, a simulation for testing the canonical EMC is still implemented, since a preliminary embedded control will be performed before moving on to a remote application. To this aim, as told in 5.2, it is sufficient to provide a constant sampling time in order to make the designed Asynchronous-EMC to behave as a normal EMC.
Then the real motor control will follow, first carried out directly on the board and then remotely, highlighting, in the latter case, both the methods of communication described in chapter 3. Before starting with the results review, three time intervals must be defined:

- T_a that is the time period with which the command is provided to the motors and it is fixed to $T_a = 20\,ms$;

- T_s that is the time interval between two measurements readings performed on the encoder;

- T_i which is the i-th time elapsed from two consecutive measurement arrivals in the asynchronous measurements and commands case; it also can be considered as the time interval between two executions of the control unit.

Please note that, in this thesis, T_i is always referenced as "sampling time" but, strictly speaking, the only sampling time is T_s; nevertheless it is not totally incorrect to define it as a sampling time, since T_i can be considered as the sampling

time from the point of view of the server, which reads the measurements with a cadence imposed by T_i itself. In addition, in case of synchronous commands and measurements, $T_i = T_s$ holds. Therefore let's to specify that whenever we speak about "sampling time", we always refer to T_i, differently, if we have to refer to T_s, it will be expressed explicitly.

6.1 Motor-Wheel Extended Plant Simulator

Before the simulations can be performed, a discussion about the extended plant implementation is needed since its purpose is to fit as much as possible to the real motor-wheel system. What has been described in section 5.1 is considered for the SIMULINK block implementation of the system dynamics and it is represented in figure 6.1; let it be referenced to as *plant*. Please note that the parameters chosen

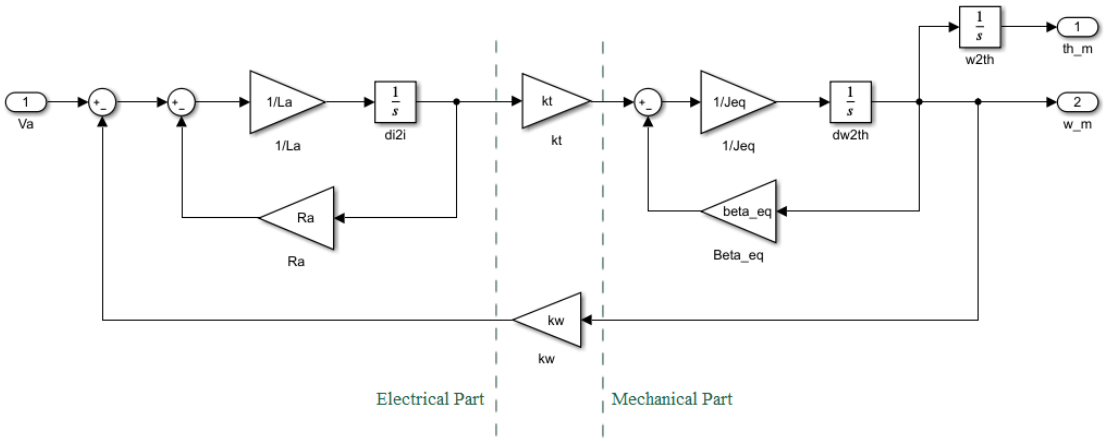


Figure 6.1: SIMULINK wheel-motor dynamics block scheme implementation

for the plant model are the same as those of the EM, this implies that between simulation and real robot application there will be a first big difference due to the fact that the parameter errors are not taken into account in this simulations.
Nevertheless, the errors regarding actuators and sensors, described in previous sections, will be considered, since they have a strong influence on the final result. Starting from the actuators, let's remind that the problems issued in 4.1.1 can be summarized in the following points:

1. minimum periodicity of $20\,ms$;
2. command voltage saturation: $-V_{max} \leq V_a \leq V_{max}$;
3. resolution error: $V_a = i\dfrac{V_{max}}{100}$, for $i \in \mathbb{Z}$;

4. dead zone for $|V_a| \leq V_{dead}$;

where V_{dead} is the minimum value in magnitude of the input voltage which makes the wheel rotate. In SIMULINK framework, these constraints are simply implemented with the block scheme represented in figure 6.2, which shows that the saturation and the resolution error are implemented by means of the appropriate blocks, while the dead-zone is implemented by means of Matlab code, where a simple if-else structure allows to set to 0 the input voltages in the dead-zone range. These three

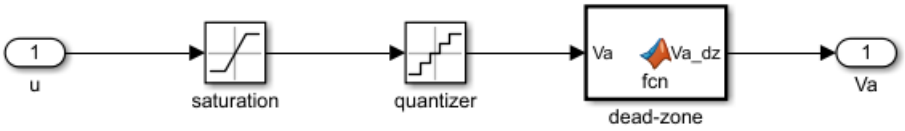


Figure 6.2: actuator implementation in SIMULINK

blocks are, then, encapsulated in a subsystem whose sampling time is set to $20\,ms$ in order to achieve the constraint on the periodicity. Figure 6.3 show the input and output of the just described subsystem.

The output of this subsystem is directly provided to the wheel-motor model,

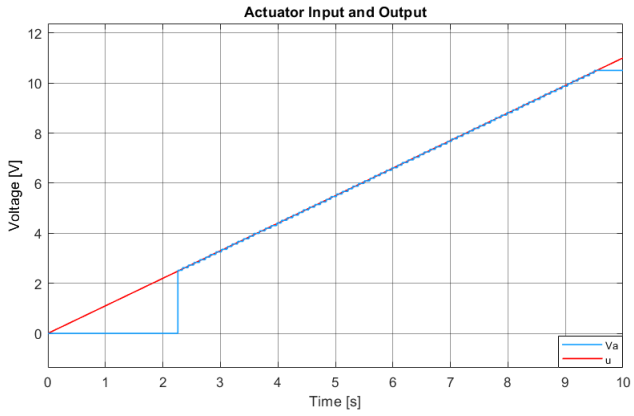


Figure 6.3: input (red) and output (blue) of the simulated actuator.

represented in figure 6.1, whose output must be handled in order to have a proper replication of the real plant behavior when it provides a measurement. Therefore a sensor model must be constructed starting from the considerations done on the two different kinds of measurements in section 4.1.2. Please remind that the direct measurement method is the one chosen for the real application, hence it is needed to implement a sensor model which emulates this method. Unfortunately it is not simple to obtain this kind of output via simulation, thus a trick is used in order to achieve the desired result. To this purpose, it is employed an observation made in

the past sections about the similarity between the direct measurements and the derived ones if the latter are filtered through the Moving Average technique and, more precisely, with a sliding window of 10 samples width.

Since all the characteristics of the encoder mounted at motor-side are well known,

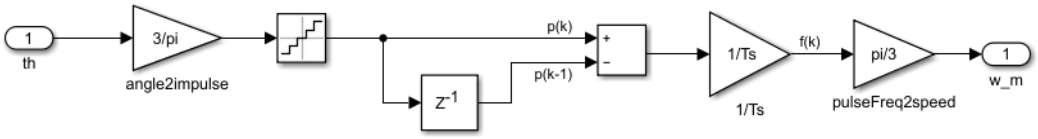


Figure 6.4: SIMULINK model for derived measurement method emulation

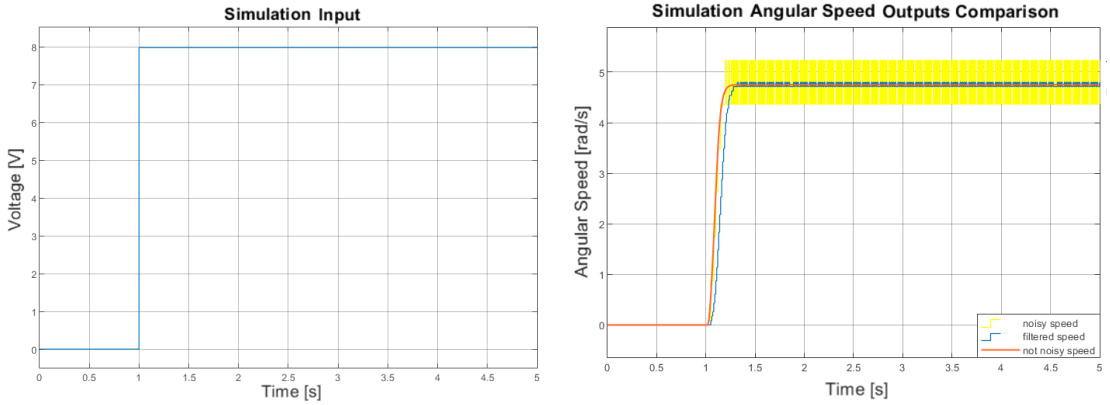
it is very simple to implement on software the emulation of the angular speed derived measurement method starting from the angular position θ_m provided by the plant model. The figure 6.4 shows a possible implementation of this sensor model whose input is the angular position which, as you can see, is divided by the encoder step angle (i.e. $\tilde{\alpha} = 60°$) in order to pass from angular position to the pulse count; the latter must be quantized since we are interested in its integer value[1]. It is worth to notice that the transition from angular position to pulse count is not mandatory, nevertheless it is performed in order to understand what sensor really do to obtain the position measurement. Then the actual pulse count must be subtracted by the previous one in order to obtain the pulse frequency by dividing the result by the sensor sampling time $T_s = 10\,ms$; finally the pulse frequency is converted in motor angular speed by multiplying the result to the encoder step angle.

As discussed previously, the final step must be the filtering of the resulting output by means of a Moving Average filter of the kind:

$$y_{filt}(k) = \frac{1}{N}\sum_{i=0}^{N-1} y(k-i), \quad y(k) = 0 : \quad \forall\, k \leq 0;$$

where $N = 10$ is the sliding window width; it is very simple to implement it using a cascade of delay blocks, whose outputs are summed and then divided by N.
In conclusion, the figure below shows the step response of the whole model which includes actuator, plant and sensor, which takes the name of *Extended Plant*. In the figure is also shown the comparison between the not-noisy output (red), taken directly from plant output, the sensor unfiltered output (yellow) and the filtered one (blue).

[1]Note that, in order to have a realistic behavior, the *quantizer* block should truncate the input, but it performs a round-to-nearest



(a) input **(b)** outputs

Figure 6.5: Extended Plant voltage input (left) and angular speed outputs (right)

6.2 EMC for Simulated Motor-Wheel System: Constant Sampling Time

This first simulation involves the application of the asynchronous-EMC in a condition of synchronous measurements and commands, providing a constant sampling time. The aim here is to test the goodness of the design of EMC components and to get an idea of the eigenvalues scale to be used in such a system. Note that, although it is not necessary since a constant sampling time is used, the eigenvalues to tune are provided in CT domain for a matter of continuity with the asynchronous case. Another preamble to make concerns the eigenvalues of the reference generator, which will be set to:

$$\lambda_{ref1}^{CT} = \lambda_{ref2}^{CT} = -3$$

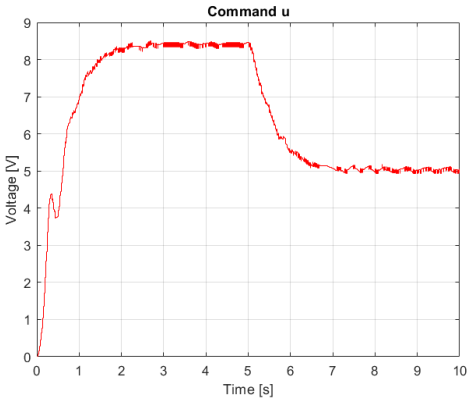
since these value allow to have a good compromise between response speed and dead-zone problem handling, which is observed in the initial phase of the control action. Nevertheless the other two sets of eigenvalues will be tuned and the related results will be shown. The following table displays the CT domain eigenvalues used in each simulation.

The following simulations will be performed for $t_f = 10\,s$ with $T_s = 10\,ms$ and the operator reference r is defined as follows:

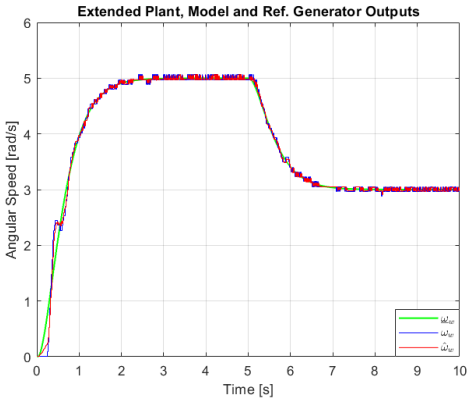
$$r(t) = \begin{cases} 5 & : 0 \leq t \leq \dfrac{t_f}{2} \\ 3 & : \dfrac{t_f}{2} < t \leq t_f \end{cases} \tag{6.1}$$

	Observer				**Controller**	
	λ_{m1}^{CT}	λ_{m2}^{CT}	λ_{m3}^{CT}	λ_{m4}^{CT}	λ_{c1}^{CT}	λ_{c2}^{CT}
simulation 1	-100	-100	-100	-100	-10	-10
simulation 2	-500	-50	-50	-50	-10	-10
simulation 3	-500	-50	-10	-60	-10	-5

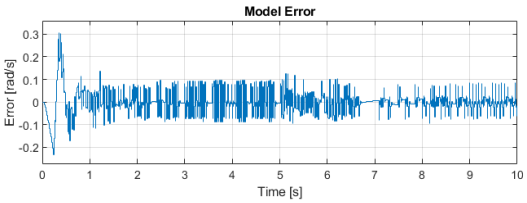
Table 6.1: EMC Observer and Controller eigenvalues for motor-wheel simulations



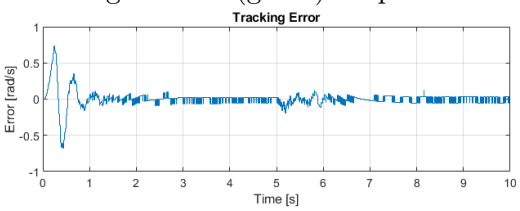
(a) controller command output u



(b) extended plant (blue), EM (red) and reference generator (green) outputs



(c) model error $\bar{e}$

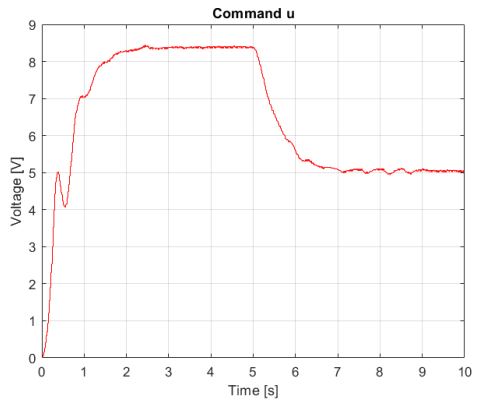


(d) tracking error $\underline{e}$

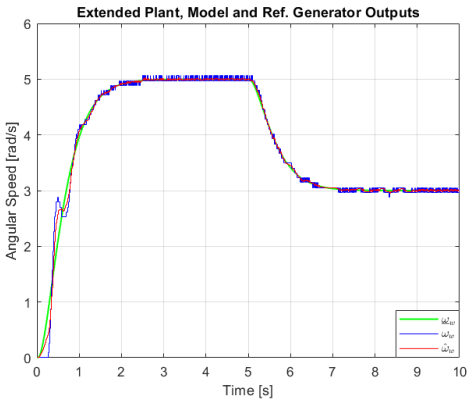
Figure 6.6: simulation 1 results

6.2.1 Simulation 1

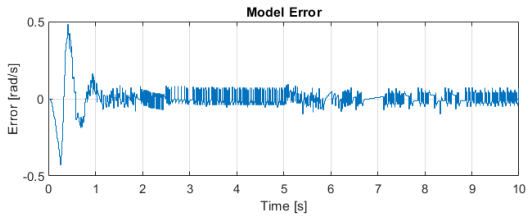
The eigenvalues choice is based on the rule of thumb for which the observer's eigenvalues must be 5 to 10 times faster than controller ones. The results in Fig. 6.6 show a small initial oscillation due to a high command action which leads to an overshoot in the starting phase of the output rise. Another problem highlighted in the figure is the fact that the estimated wheel angular speed includes a too high frequency dynamic which must be filtered.



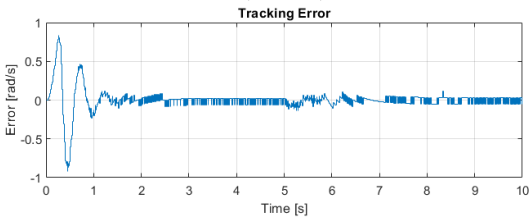
(a) controller command output u



(b) extended plant (blue), EM (red) and reference generator (green) outputs



(c) model error $\bar{e}$



(d) tracking error $\underline{e}$

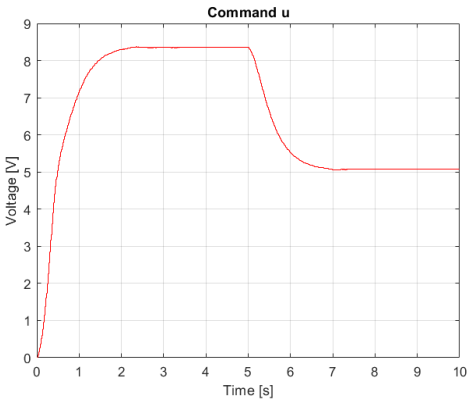
Figure 6.7: simulation 2 results

6.2.2 Simulation 2

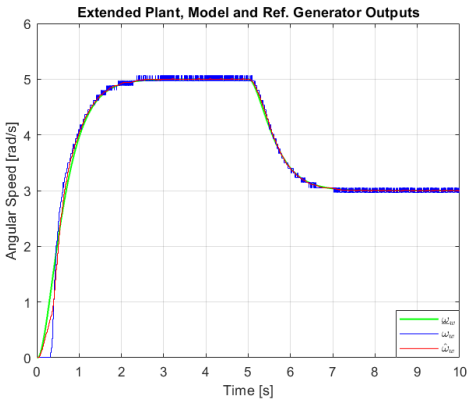
The observer eigenvalue related to the current is increased since the electric dynamics are certainly more rapid than the mechanical ones. On the other hand the other three observer eigenvalues are decreased since the aim is to cut off the high frequency noises from the final estimate of the speed state. The results (see Fig. 6.7) show that the filter action is achieved but the initial oscillation is still too pronounced.

6.2.3 Simulation 3

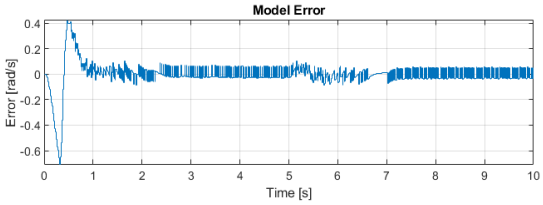
In order to obtain a better result on the rising phase, the control eigenvalue related to the rotational speed must be decreased, this leads to a lower control action and, then, a lower overshoot. In Fig. 6.8 it can be noticed that these eigenvalues provide a filtered estimation, in addition the previously mentioned overshoot is well damped, even if it doesn't completely disappear.



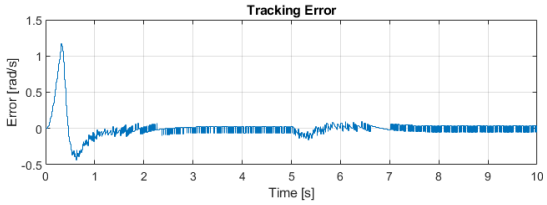
(a) controller command output u



(b) extended plant (blue), EM (red) and reference generator (green) outputs



(c) model error $\bar{e}$



(d) tracking error $\underline{e}$

Figure 6.8: simulation 3 results

6.2.4 Results summary

Focusing on the resulting error plots it is possible to highlight two fundamental aspects which are common to all the simulations:

- for the first 2 seconds an oscillation can be noted for both types of error. As anticipated earlier, this behavior is due to the dead-zone introduced by the actuator, discussed in section 4.1.1, which causes a disproportionate command action that results in an initial overshoot. The speech above is proved by the fact that no oscillation is found in the falling edge within the time interval from $5\,s$ to $7\,s$, in which the output ranges from $6\,rad/s$ to $4\,rad/s$; in fact, in this speed range, the voltage input works outside the dead-zone. In conclusion, it can be understood that this problem will be present in all the next results, nevertheless, with the current design of the EMC, the only tool to counter it is the tuning of the eigenvalues, in particular that of the reference generator: by slowing down them, it is possible to have a smoother and slower reference which would make the command action less aggressive at the moment of detachment from the dead-zone.

- Apart from these first two seconds, the errors always remain in very low margins: in the last simulation, for example, both the model and tracking errors are within the range $[-0.2,\, 0.2]\, rad/s$ from the instant $t' = 2\, s$ on. In conclusion, it can be stated that these errors are mainly attributable to the disturbance in the measurements.

6.3 EMC for Simulated Motor-Wheel System: Constant Sampling Time

At this point we move on to the simulation of a system with asynchronous measurements and commands, testing the control with a variable sampling time. The purpose of this section, differently from the work done in the previous one, is not to expose an eigenvalues tuning, as it could be not very interesting, but to make understand how much the variability of the sampling time can affect the final result. Anyway, before deepen this topic, a brief description of the tools used to carry out such a simulation is needed.

6.3.1 Networked Control System simulation

In order to simulate a system of asynchronous measurements and commands on the SIMULINK framework, it is necessary to use two blocks already presented in the section 2.4.1, namely the *Transport Delay* and the *Triggered Subsystem* blocks. As already explained in that section, their combined usage allows to emulate the transmission delay and the delay variability respectively, both caused by the insertion of a communication network (CN) in the closed loop.
About *Transport Delay* simply is a buffer which delays the input signal by a specified time. In this case, by using the results of 3.2.3, it is used a delay time of $t_d = 8.5\, ms$. On the other hand, the *Triggered Subsystem* block may be of greater interest since it is the one that allows to simulate a variable sampling time. Please remember that this block is basically a subsystem that is executed at each rising edge of a square wave signal that is given to it as an input. In order to have a simulation that is as realistic as possible, it is needed to have a sampling time which varies in a random fashion and with a proper variance. To this aim it is needed to construct a square wave with a variable period which must be provided to the *Triggered Subsystem*, as shown in the figure 6.9 where the block *Variable Pulse Generator* is the one in charge for this task. The most important role is played by the *Random Number* block which generates random values with a Normal distribution to which must be specified the mean value μ and the variance σ^2 of the stochastic process. In this particular case it holds $\mu = T_s = 10\, ms$, as a consequence of the fact that the plant sends a measurement each T_s, while σ^2 is a parameter which will be changed in the

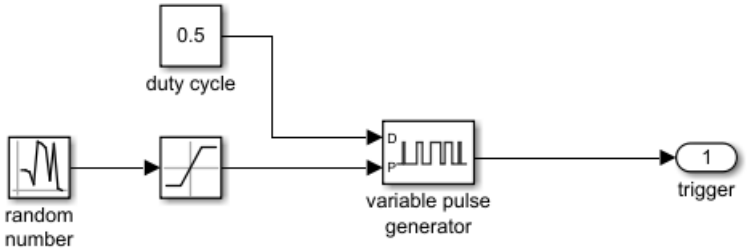


Figure 6.9: SIMULINK block scheme of square wave generator for Triggered Subsystem block.

following simulations in order to prize the effect of the sampling time variability to the controlled system. Finally, the random numbers generated by this block, are provided to the pulse generator, adjusting the time interval between two rising fronts of its output square wave, nevertheless not all values are admitted as input for pulse generator thus it is needed a saturation block which prevents this input from assuming values below $1\,ms$ and above $100\,ms$, as shown in figure.
In conclusion, the simulations whose results will be shown below differ from each other for the variance of the sampling times, which, for simplicity of notation, will be indicated with its standard deviation σ. In addition, remember that the final result is also influenced by the stochastic process realization of the random number generator, which changes with the variation of the seed that is supplied as a parameter to the block. For this reason, each simulation presents 4 of the most representative results, each obtained by supplying a different seed to the random number generator. Moreover the eigenvalues used for all the simulations are the same and are exposed in the table 6.2 and the operator reference defined in 6.1 is used.

Observer				**Controller**		**Reference**	
λ_{m1}^{CT}	λ_{m2}^{CT}	λ_{m3}^{CT}	λ_{m4}^{CT}	λ_{c1}^{CT}	λ_{c2}^{CT}	λ_{ref1}^{CT}	λ_{ref2}^{CT}
-100	-70	-5	-60	-70	-60	-2	-2

Table 6.2: EMC Observer, Controller and Reference Generator eigenvalues

6.3.2 Simulation 1: $\sigma = 5\,\text{ms}$

The figure 6.10 shows the results of the simulation performed with $\sigma = 5\,ms$, it immediately can be seen that there are no particular changes between the realizations. The errors are very good, in fact they only show the usual little oscillation in the initial phases which, however, is immediately dampened, bringing both errors to remain stably in very low values.

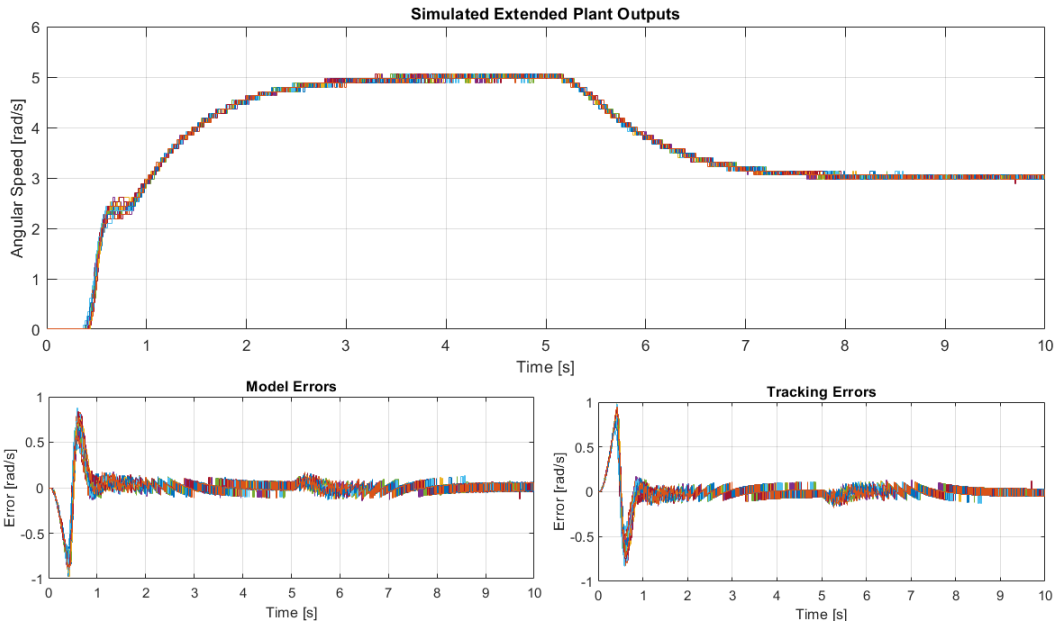


Figure 6.10: simulation 1: extended plant outputs (top), model errors (bottom left) and tracking errors (bottom right) in 30 different realizations of T_i, with $\sigma = 5\,ms$.

6.3.3 Simulation 2: $\sigma = 10\,\text{ms}$

The figure 6.11 shows the results of the second simulation, performed with $\sigma = 10\,ms$. Here the effects of a greater variability of sampling times is much evident then the previous simulation, especially in the realization represented by with the light blue line, which is the one with a more pronounced oscillation on the rising phase. It can be observed that the difference between realizations is more visible with respect to the simulation 1. Nevertheless, also for the simulation 2, the error is limited to the first phase where the dead-zone strongly affect the performance of the controller in the first two seconds.

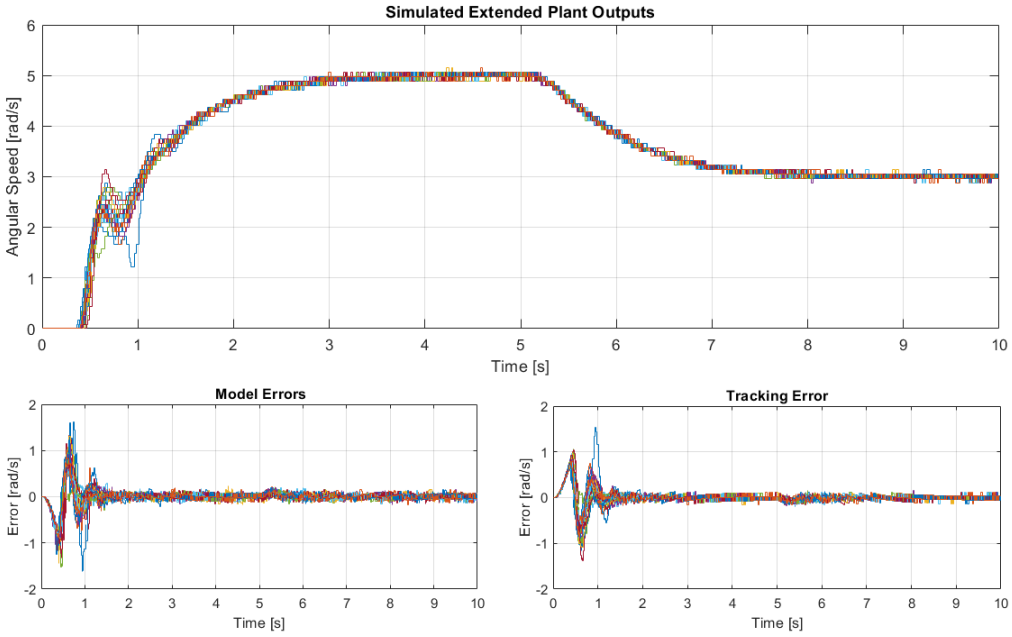


Figure 6.11: simulation 2: extended plant outputs (top), model errors (bottom left) and tracking errors (bottom right) in 30 different realizations of T_i, with $\sigma = 10\,ms$.

6.3.4 Simulation 3: $\sigma = 15$ ms

The simulation 3 is obtained with a standard deviation $\sigma = 15\,ms$. Here the control is very deteriorated, in fact, how it can be seen in figure 6.12, there are some realizations (e.g. the light blue line) where the oscillations are not limited to the first rising phase but they are extended to the whole output. Anyway, although the oscillations, the system remains stable, showing a good robustness to the sampling time variations. Of course, by augmenting more and more σ, instability can occur.

6.3.5 Results summary

Before commenting these results, let's clarify how the choice of the variance was performed. Please remind that, in the Normal distributions, the values included in the interval $[\mu - 3\sigma, \mu + 3\sigma]$ have an outcome probability of more than 99%; therefore it is possible to get the standard deviation by defining a desired range of values within which there must lie the most of the sampling times values.
At this point, the results show that the controlled system starts to reveal problems when the variability of the sampling times rises, and this can be explained by the fact that the model isn't able to follow the plant since plant and model are

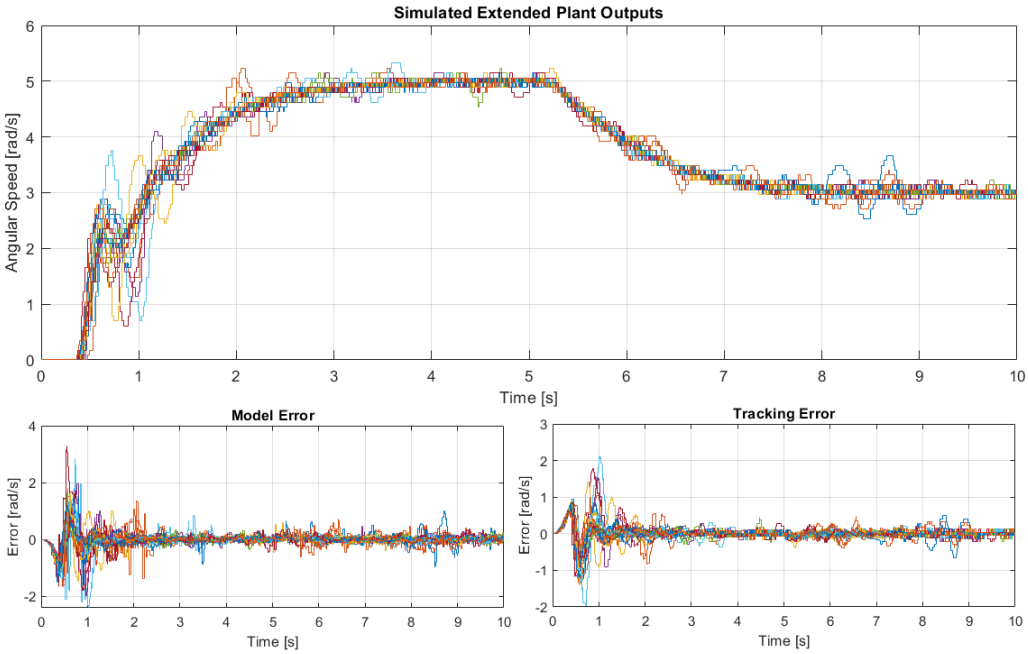


Figure 6.12: simulation 3: extended plant outputs (top), model errors (bottom left) and tracking errors (bottom right) in 30 different realizations of T_i, with $\sigma = 15\,ms$.

totally out of sync. Nevertheless, it is worth to notice that the simulation has a big difference with respect to the real system: supposing that there is no transmission delay (i.e. the Transport Delay block is neglected), in the simulation the sample obtained at the instant t_i from the block of the control unit is a measurement read in that same instant by the extended plant, therefore all the measurements performed on the plant between two consecutive samples can be considered to be lost. On the other hand, in a real network communication, two samples arriving to the server at two consecutive instants are certainly measurements performed in two consecutively on the robot, since it is assumed that there is neither loss nor exchange of packets in the transmission. Note that this constraint also imposes a certain rationality on the variability of sampling times, for example, if a sample arrives after $100\,ms$, it is normal to expect subsequent samples arrivals at shorter time intervals, since Network mechanisms, such as buffering, come into play. Unfortunately all these aspects can't be easily implemented with a simulation, thus differences between simulations and real system tests will be experienced.
Anyway, in terms of control performance, all what was highlighted in section 6.2.4 is confirmed here, in fact, looking at the errors of all three simulations, it can be seen that the most problematic part is always the rising phase, in which the

oscillations are more pronounced than elsewhere. Even in the last simulation, which is characterised by an high variance, there is one stochastic realization in which oscillations are experienced also after the rising phase, but, also in this case, both the errors remain restrained to low values.

Chapter 7

Wheels Rotational Speed Control Experimental Tests

In this chapter the goal is to show the results of the experimental tests performed on the designed EMC applied to the real system, both in the on-board application and in the remote one. To this aim it is used a `C++` project which includes a program executed on the robot board and, for the case of remote control, a program executed on the server. The details of the project are not deepened but some outlines are provided in the following sections. Nevertheless it is fundamental to underline that it is possible to use exactly the same EMC implemented in the simulator on the `C++` project too. This can be done by using *Code Generator* of SIMULINK that is a tool which allows to generate a code starting from a target *Subsystem* block. Moreover, since the generated code implements a `C++` class which replicates a SIMULINK subsystem, it provides methods for the access to inputs, states and outputs of the converted subsystem ant it also provides a method for a single step execution, called *step()*. In this specific context the target subsystem is the one which implements the whole EMC, this means that, by referring to the scheme in figure 2.1, the inputs to provide to the class are the measurement y and the operator reference r, while the output generated after a *step()* call is the command u. In conclusion, when it comes to the on board control, this piece of code must be executed on the Raspberry, on the other hand, when a remote control is tested, it must be executed on the server.

7.1 EMC for Real Motor-Wheel System: On Board Control

In the project to be executed on the robot board, actuators and sensors are managed through libraries provided by the manufacturers (see section 4.2), while the EMC is managed by a class automatically generated by the SIMULINK tool mentioned above. Therefore the rest of the project becomes relatively simple to implement, in fact it is only a matter of being able to give a specific timing to the actions to perform. More in detail, to keep a continuity with the simulations shown previously, the use of two timers is needed, one that expires every $T_s = 10\,ms$ and which gives the timing for both the measurement and the execution of the function *step()*, while the other timer expires each $T_a = 20\,ms$ and gives the timing for the actuation of the motors.

7.1.1 Wheels on Board Control Tests General Settings

The goal of the tests is not to show the process of eigenvalues tuning, in fact it is more interesting to see what is the response of right and left controlled wheels wrt the operator reference r. So the eigenvalues used for these tests are are assumed to be fixed and the tables 7.1 and 7.2 point out the values used for the EMC eigenvalues of the right wheel and the left wheel respectively.

Please note that the two tables show different eigenvalues, this means that they

Right Wheel EMC Eigenvalues							
Observer				**Controller**		**Reference**	
λ_{m1}^{CT}	λ_{m2}^{CT}	λ_{m3}^{CT}	λ_{m4}^{CT}	λ_{c1}^{CT}	λ_{c2}^{CT}	λ_{ref1}^{CT}	λ_{ref2}^{CT}
-100	-50	-5	-60	-10	-5	-2	-2

Table 7.1: EMC Observer, Controller and Reference Generator eigenvalues for right wheel speed control tests

can potentially react with different responses even by providing to the wheels the same operator reference to track. Anyway the two motor-wheel systems have some structural differences which prevent them to have the same response even using the same eigenvalues.

In the figures 7.4 and 7.3 the step responses of both the wheels are displayed and, in this preliminary test, the eigenvalues of the table 7.1 are provided to both the EMC. It can be observed that the two outputs are not exactly the same and this proves what said above. Therefore, in order to ensure that both wheels

Left Wheel EMC Eigenvalues							
Observer				**Controller**		**Reference**	
λ_{m1}^{CT}	λ_{m2}^{CT}	λ_{m3}^{CT}	λ_{m4}^{CT}	λ_{c1}^{CT}	λ_{c2}^{CT}	λ_{ref1}^{CT}	λ_{ref2}^{CT}
-1000	-35	-10	-60	-200	-5	-2	-2

Table 7.2: EMC Observer, Controller and Reference Generator eigenvalues for left wheel speed control tests

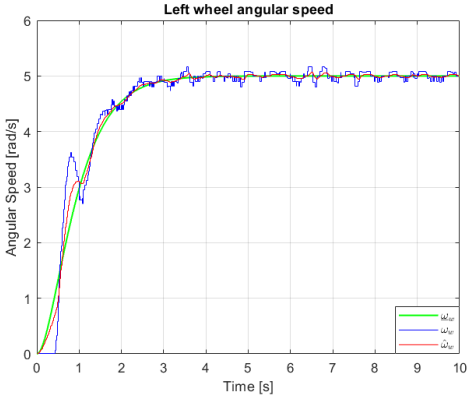


Figure 7.1: left wheel, step response with eigenvalues in Tab. 7.1

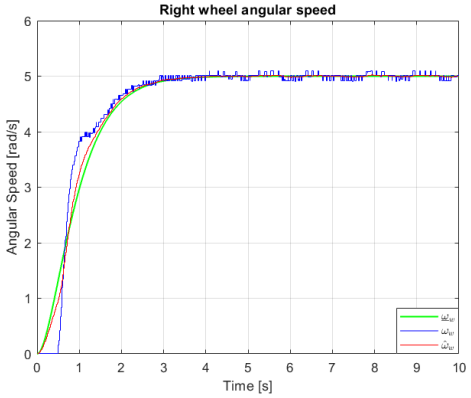
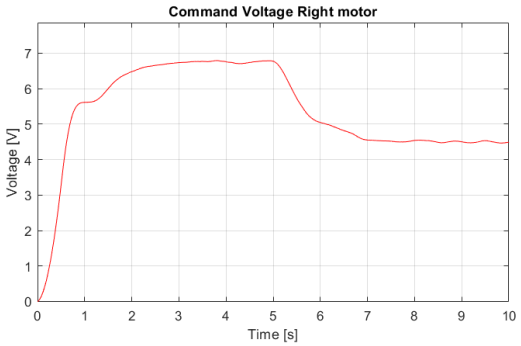


Figure 7.2: right wheel, step response with eigenvalues in Tab. 7.1

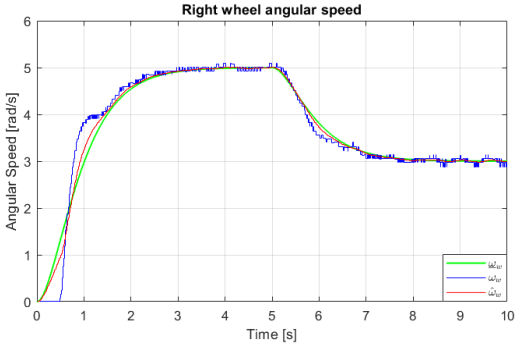
provide a similar response, it is needed that both of them are able to follow the respective reference y with the minimal tracking error possible; obviously the other necessary condition is that reference generators of right and left wheels have the same eigenvalues, because the angular speed references must be the same.
Then it is worth to notice that the reference generator eigenvalues are decreased since it is needed a slower dynamic for the reference because, in real case, the detachment from the dead-zone is more aggressive wrt what was seen in the simulations. At this point, starting from this basic settings, some of the most significant tests, performed on both the wheels, are shown in the following.

7.1.2 Wheels on board control: test 1

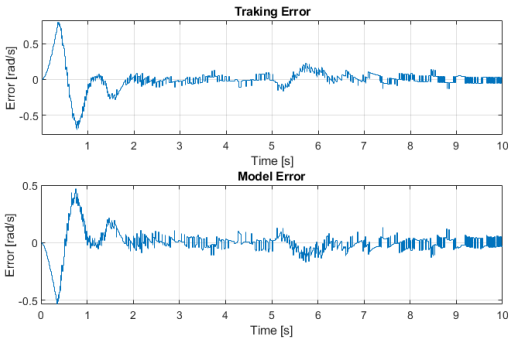
The goal of this test is to verify the goodness of the control system applied to both the wheels. To do this aim the reference defined in 6.1 is provided to both the wheels, where the duration time of the test is $t_f = 10\,s$. The figure 7.3 shows the results of this test for the right wheel while the figure 7.4 shows the results of the EMC applied to the left wheel.



(a) controller command output u

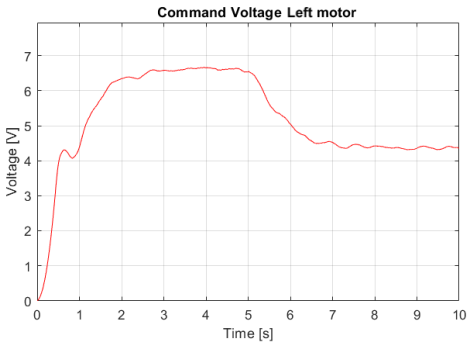


(b) measurements (blue), estimate (red) and reference (green)

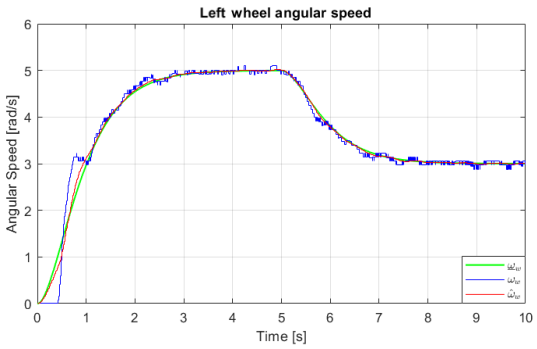


(c) model error $\bar{e}$ (bottom) and tracking error $\underline{e}$ (top)

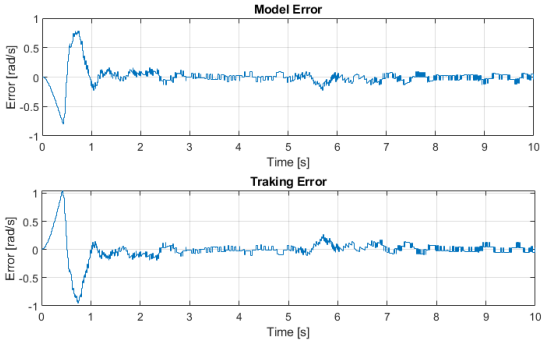
Figure 7.3: right motor test 1 results



(a) controller command output u



(b) measurements (blue), estimate (red) reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

Figure 7.4: left motor test 1 results

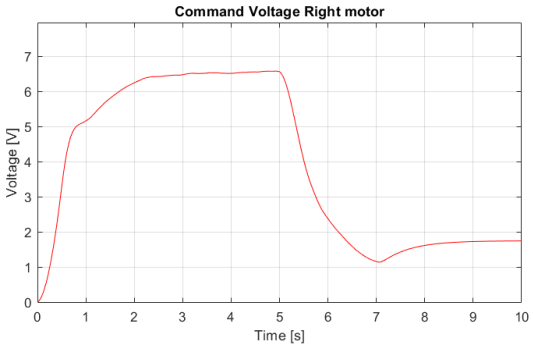
7.1.3 Wheels on board control: test 2

In this second test the reference provided to the wheels is a port function as the one written below.

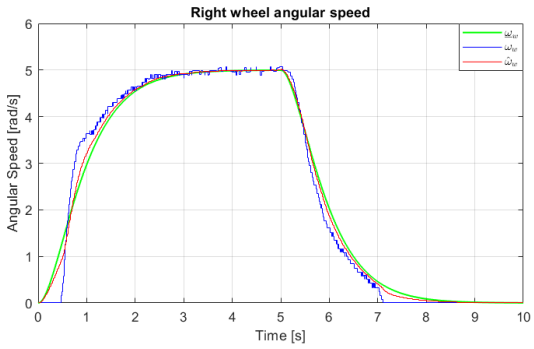
$$r(t) = \begin{cases} 5 & : \ 0 \leq t \leq \dfrac{t_f}{2} \\ 0 & : \ \dfrac{t_f}{2} < t \leq t_f \end{cases} \tag{7.1}$$

Practically it is a wheel braking test and it is very interesting since it implies that the input voltage must return to the dead-zone, potentially causing some problems in the control system. Figures 7.5 and 7.6 represent the results of the test of right wheel and the left wheel respectively.

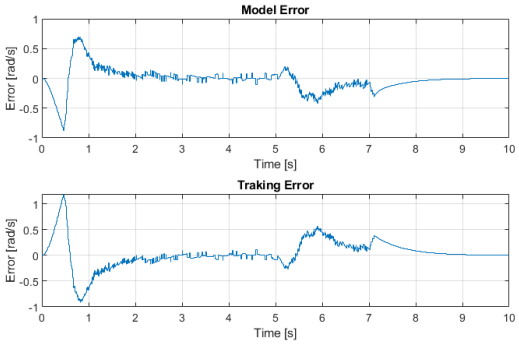
As you can see, around the instant $t = 7\,s$ there is an immediate drop to zero



(a) controller command output u



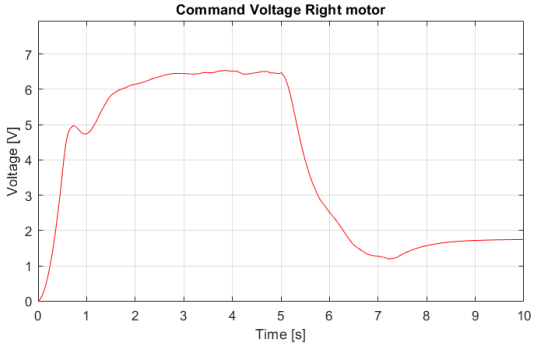
(b) measurements (blue), estimate (red) and reference (green)



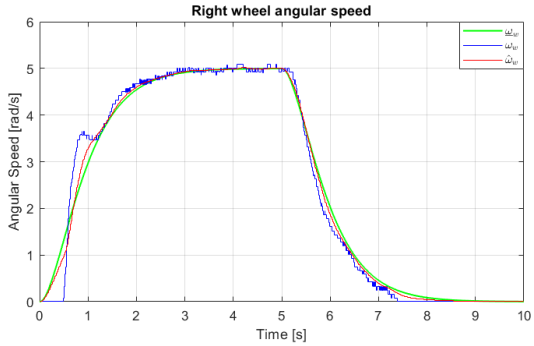
(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

Figure 7.5: right motor test 2 results

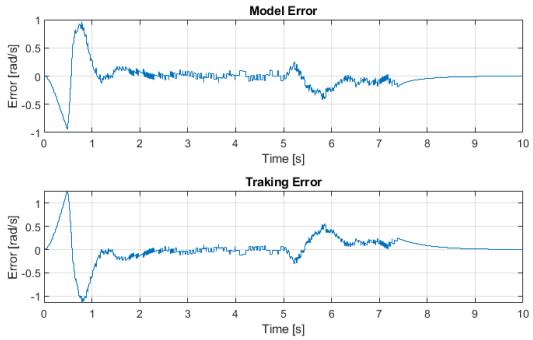
of the measurements on both wheels, pointing out the entry to the dead-zone. It is also worth to notice that the command u settles on a non-null value, this is



(a) controller command output u



(b) measurements (blue), estimate (red) reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

Figure 7.6: left motor test 2 results

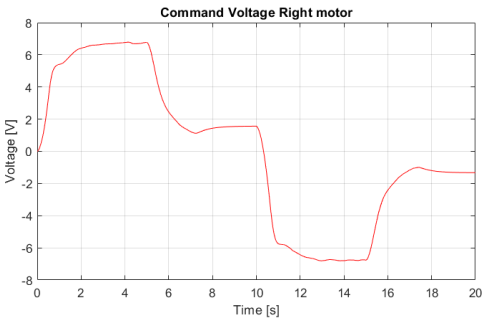
because, according to EMC, it is the input value that makes the wheels stop. In reality we know that there is a range of values that brings the output speed to zero (i.e. dead-zone), but the control system simply settles on the first value it meets in this range, which is in particular $u_1 = 1.74\,V$, making to note that, when the wheel is in running, the dead-zone has a smaller range of values than that indicated in section 4.1.1.

7.1.4 Wheels on board control: test 3

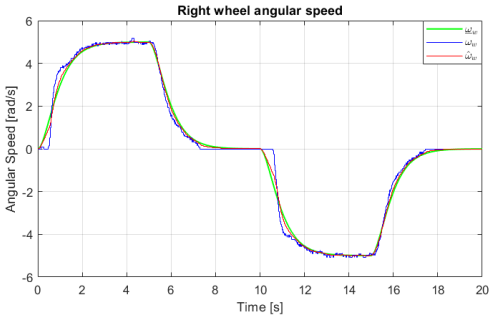
The third and last test is made to provide more evidences to support the findings of the last test. In fact in this case a double port reference is given as input to both the wheels, in order to study the behavior of the systems when the dead-zone is approached starting from negative speed values.
Thus let be

$$r(t) = \begin{cases} 5 & : 0 \leq t \leq \dfrac{t_f}{4} \\ 0 & : \dfrac{t_f}{4} < t \leq \dfrac{t_f}{2} \\ -5 & : \dfrac{t_f}{2} < t \leq \dfrac{3\,t_f}{4} \\ 0 & : \dfrac{3\,t_f}{4} < t \leq t_f \end{cases} \tag{7.2}$$

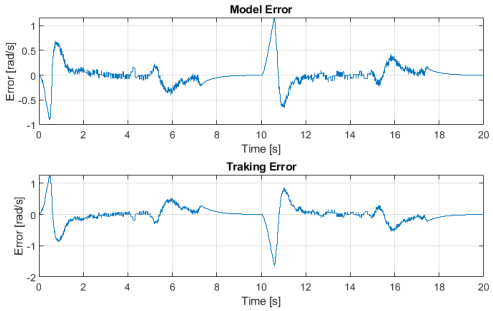
the operator reference provided to the wheels, where $t_f = 20\,s$, then the results are exposed in figures 7.7 and 7.8



(a) controller command output u

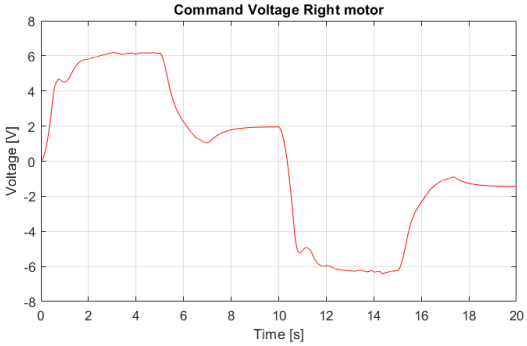


(b) measurements (blue), estimate (red) and reference (green)

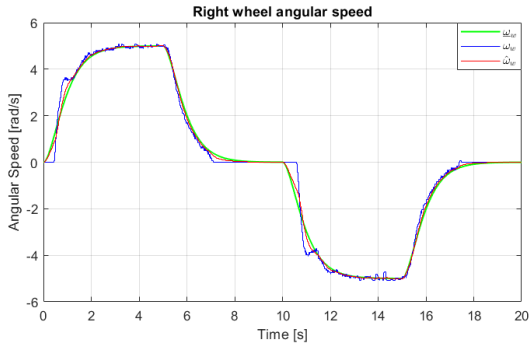


(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

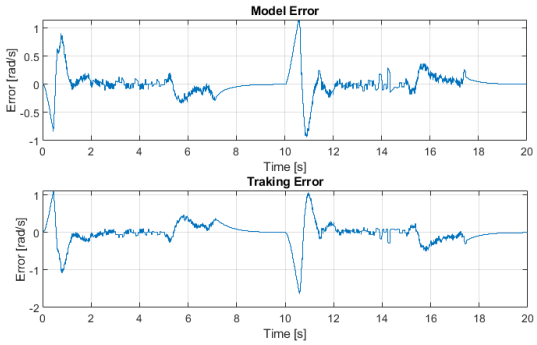
Figure 7.7: right motor test 3 results



(a) controller command output u



(b) measurements (blue), estimate (red) reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

Figure 7.8: left motor test 3 results

7.1.5 Results summary

With this tests the goal is to provide proofs of the validity of the designed control system applied to the real robot wheels. The first result that does not stand out from these figures, but which is worth pointing out, is that the parametric errors mentioned in section 5.1.1 are rejected by the EMC, exactly as expected from the theory. The second result concerns the management of the non-linearity inserted by the dead-zone in the plant. Especially the last test points out that the dead-zone varies from engine to engine and, in the same engine, it varies in the direction of rotation speed of the wheel. This type of behavior means that, whenever the wheel approaches the dead-zone, both the model error and the tracking error increase. Nevertheless, on the whole, the control system still responds very well, although the transient can be influenced by the dead-zone, the final target is still reached relatively shortly, in fact in 2.5 s both wheels reach 96% of the final target.

7.2 EMC for Real Motor-Wheel System: Remote Control

Finally the tests of the remote control applied to the wheels can be performed, first by using the p2p connection between laptop and robot board, and then by employing a router to forward the UDP packets which the two hosts exchange.

7.2.1 Wheels Remote Control Tests General Settings

Regardless of which of the two communication methods is used, the general setting of the project is the following:

- the *client*, implemented on the robot board, is in charge of performing measurements and sending them to the server at a cadence of $T_s = 10\,ms$, and in the meanwhile it must actuate motor at a cadence of $T_a = 20\,ms$. In addition a second thread is implemented to read the packets arriving from the server;

- the *server* is implemented on the laptop and, also in this case, two threads are employed: one for reading the measurements sent by the client and the second one used to execute the EMC and to send back the resulting command to the client.

Thus from one network architecture to another only the methods to send and receive packets change in software implementation. The following tests will show a comparison between the control performances in the two communication methods, using eigenvalues that are different from wheel to wheel (as in the previous test) but that do not change between one test and another. The tables 7.3 and 7.4 show the eigenvalues used for the remote control of the right wheel and left wheel respectively. In addition let's specify that a limit of the sampling times is needed

Right Wheel EMC Eigenvalues							
Observer				**Controller**		**Reference**	
λ_{m1}^{CT}	λ_{m2}^{CT}	λ_{m3}^{CT}	λ_{m4}^{CT}	λ_{c1}^{CT}	λ_{c2}^{CT}	λ_{ref1}^{CT}	λ_{ref2}^{CT}
-100	-65	-2	-60	-80	-15	-2	-2

Table 7.3: EMC Observer, Controller and Reference Generator eigenvalues for right wheel speed remote control tests

since either too high or too low sampling times can cause instability in the control loop. Obviously this limitation is imposed merely for the software layer, it is

Left Wheel EMC Eigenvalues							
Observer				**Controller**		**Reference**	
λ_{m1}^{CT}	λ_{m2}^{CT}	λ_{m3}^{CT}	λ_{m4}^{CT}	λ_{c1}^{CT}	λ_{c2}^{CT}	λ_{ref1}^{CT}	λ_{ref2}^{CT}
-100	-60	-1	-60	-80	-12	-2	-2

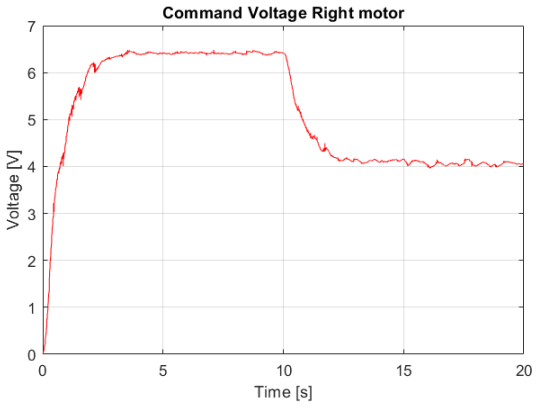
Table 7.4: EMC Observer, Controller and Reference Generator eigenvalues for left wheel speed remote control tests

impossible to limit the time intervals between two sample arrivals. In particular, for these tests, the sampling times are limited as follows: $1\,ms \leq T_i \leq 30\,ms \;\forall\; i$. Also in this case some interesting references are provided to the systems in order to obtain a comparison with the tests carried out previously with the control performed directly on the robot board.

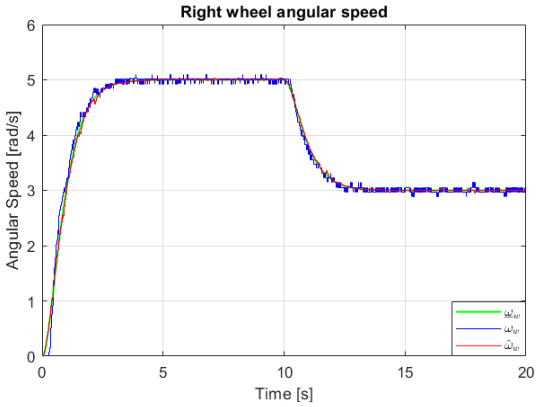
7.2.2 Wheels remote control, p2p connection: test 1

In this test the operator reference provided to the wheels is the that defined in 6.1 for the test 1 of the embedded speed control with a test time of $t_f = 20\,ms$. The results for the two wheels are shown in figures 7.9 and 7.10, where also the realization of the sampling times are reported.

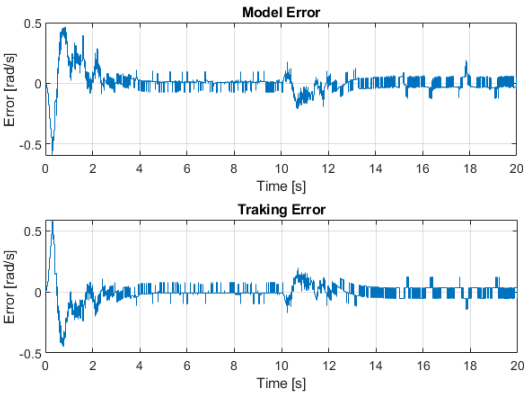
An aspect to highlight is that, as can be observed in the figures 7.10d and 7.9d, the sampling times can assume value $T_i = 0$ for some i, but these zero values are a consequence of the buffering action performed by the server before the operating system sends the received data to the user space. When this happens, more than one packet is sent to the application at once, implying that the reading times between two consecutive packets are some order smaller than the millisecond.



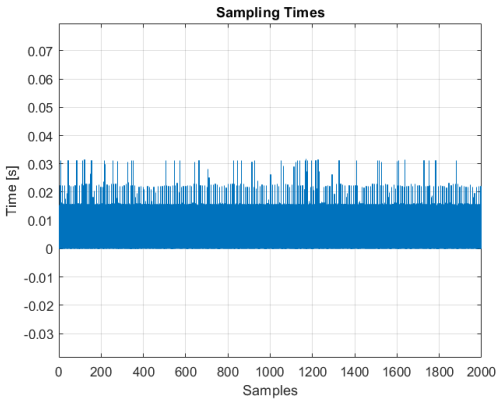
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)

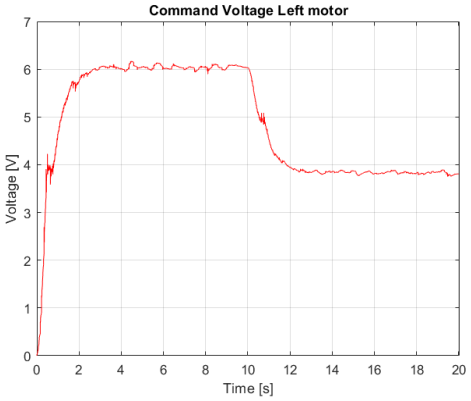


(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

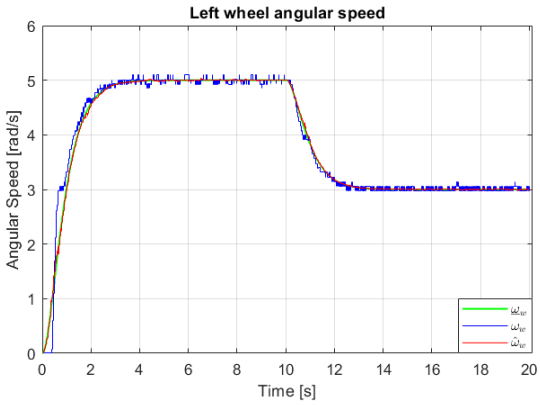


(d) sampling times

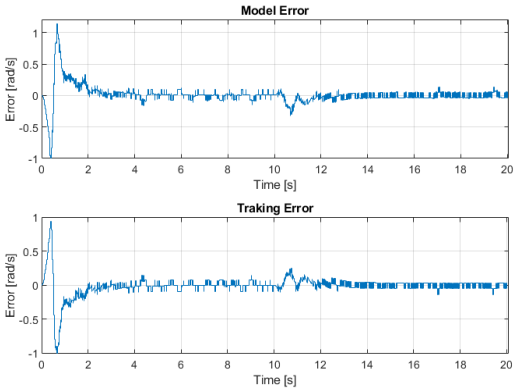
Figure 7.9: right wheel remote control, p2p connection, test 1 results



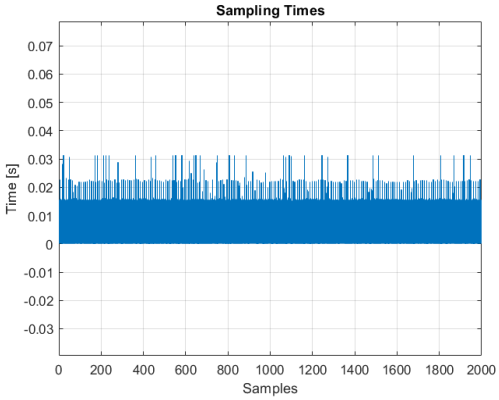
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

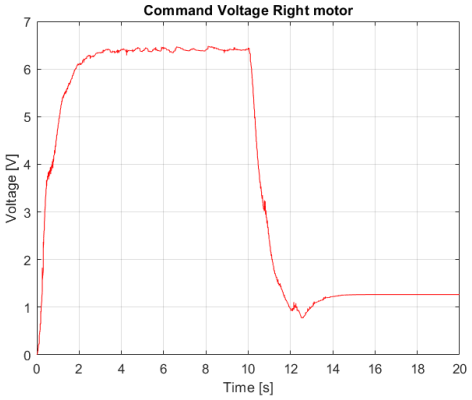


(d) sampling times

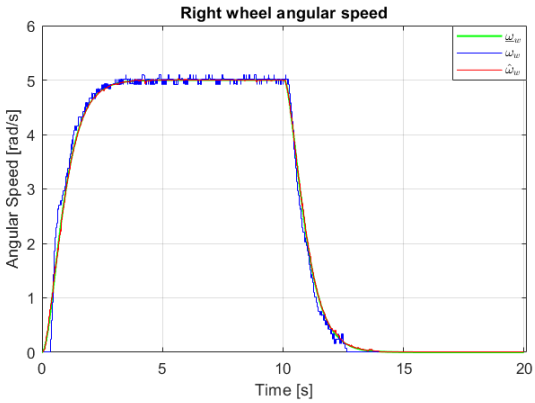
Figure 7.10: left wheel remote control, p2p connection, test 1 results

7.2.3 Wheels remote control, p2p connection: test 2

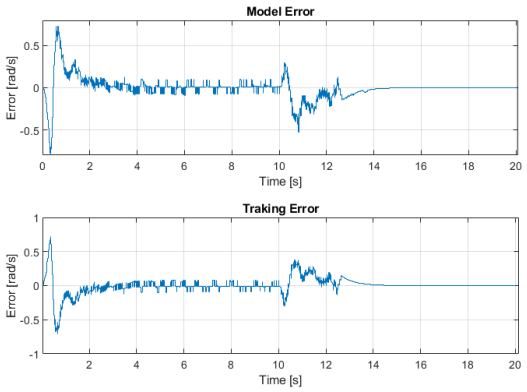
This test deals with the braking case, already studied with the control of the wheels performed directly on the board. The reference for this test is the one defined in equation 7.1 and the results are shown in figures 7.11 and 7.12 for the right and left wheel respectively. Let's highlight that, thanks to low variability of the sampling



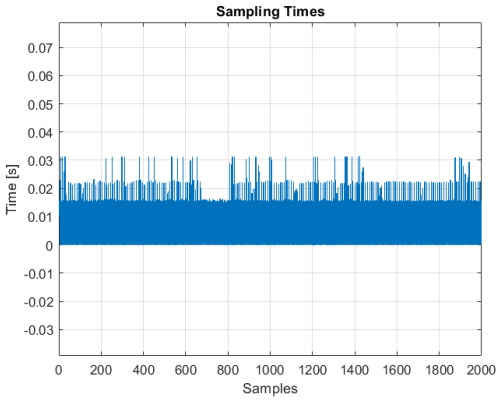
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



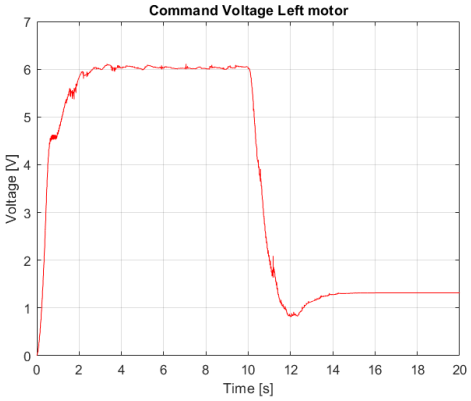
(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)



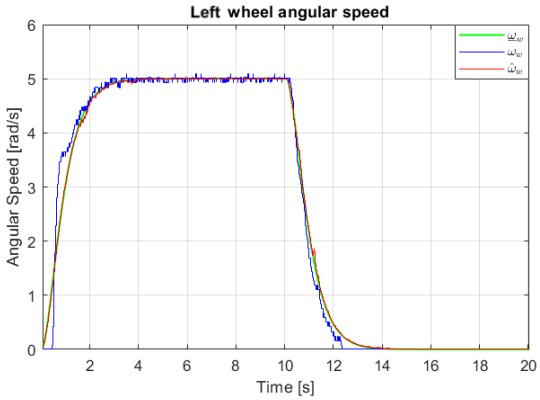
(d) sampling times

Figure 7.11: right wheel remote control, p2p connection, test 2 results

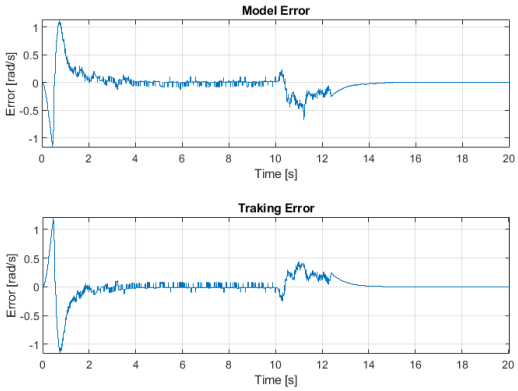
times, the results are very good and comparable to the results obtained by the on board control tests. However these tests are not as reliable, first of all because a realistic network connection is much more complex and therefore introduces more variability in sampling times; furthermore, since the physical connection is via Ethernet cable, the robot cannot stay with the wheels attached to the floor, hence the results do not take into account the friction between wheel and floor.



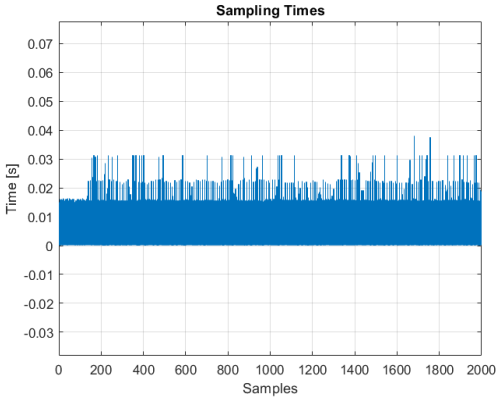
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

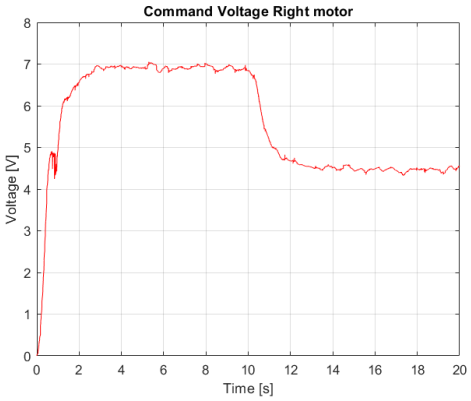


(d) sampling times

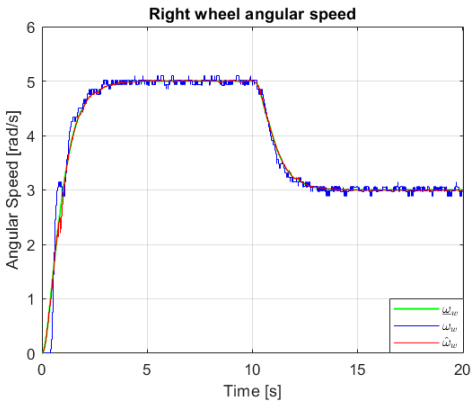
Figure 7.12: left wheel remote control, p2p connection, test 2 results

7.2.4 Wheels remote control, WiFi connection: test 1

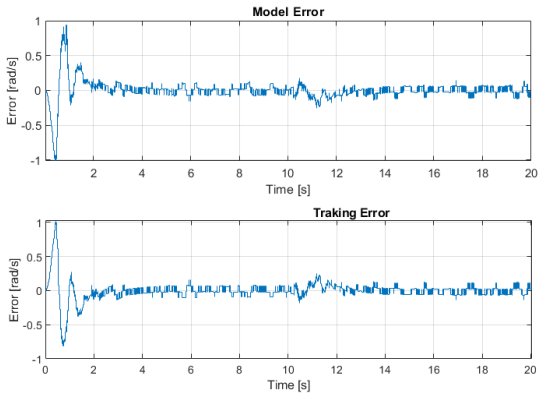
As anticipated, to carry out this test and the next, a connection between robot and server is taken into consideration which includes the use of a router to forward the packets exchanged to each other. Both devices are connected to the router via WiFi and the latter forwards the packets according to the destination UDP port it reads from the packet (a PAT configuration is needed). The reference is the same used for 7.2.2, and the results for right and left wheel are shown in figures 7.13 and 7.14 respectively.



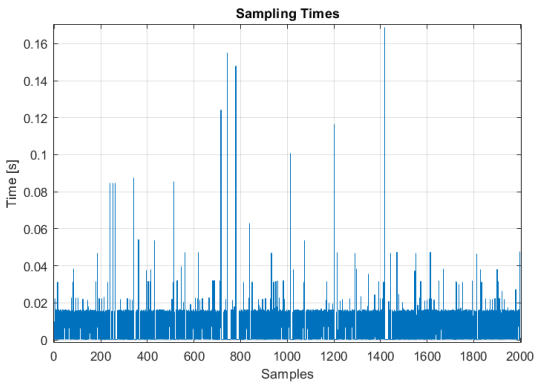
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)

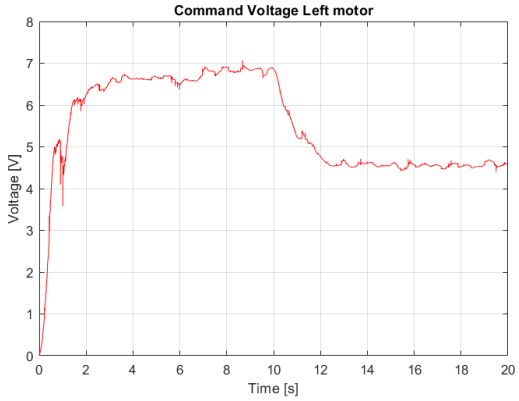


(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

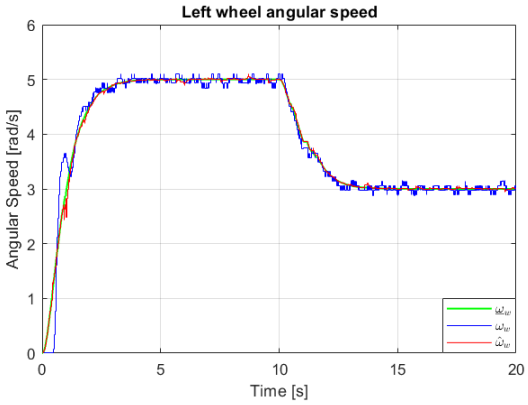


(d) sampling times

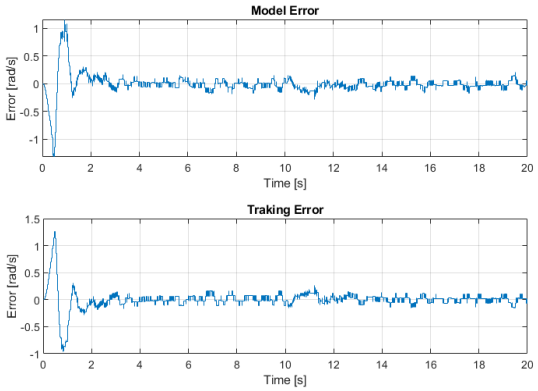
Figure 7.13: right wheel remote control, WiFi connection, test 1 results



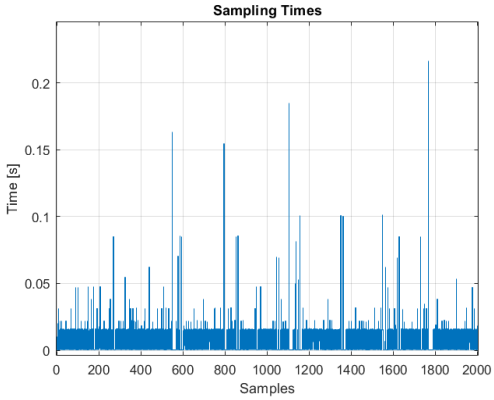
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

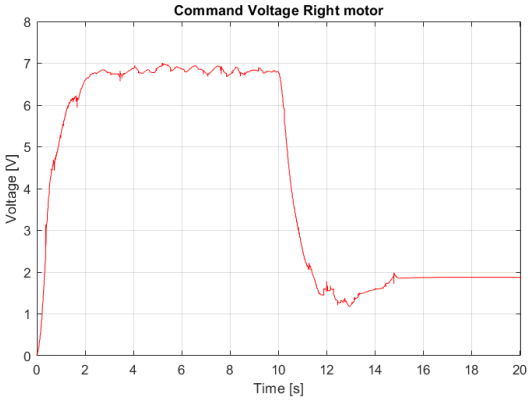


(d) sampling times

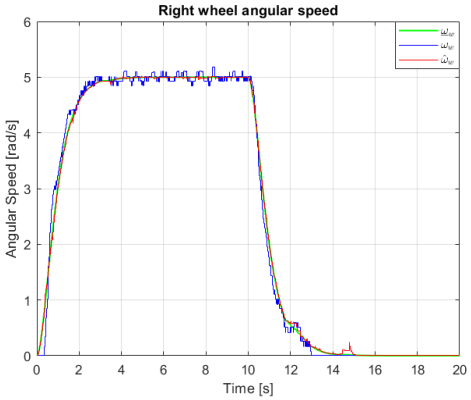
Figure 7.14: left wheel remote control, WiFi connection, test 1 results

7.2.5 Wheels remote control, WiFi connection: test 2

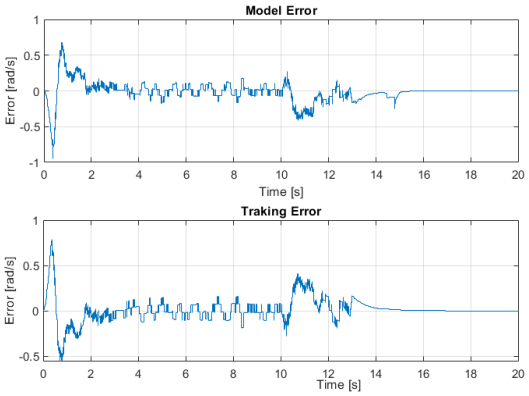
The final test performed for the remote control system is the usual braking test, whose reference is defined as 7.1. The following figures, namely 7.15 and 7.16 show the results.



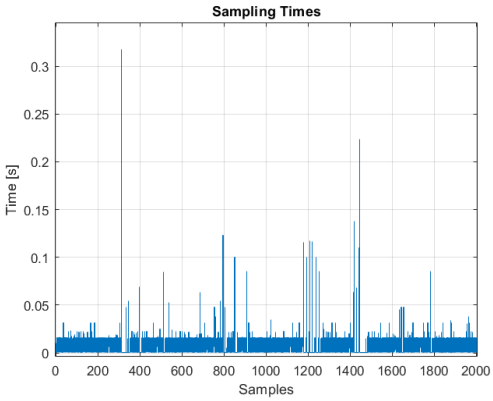
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)

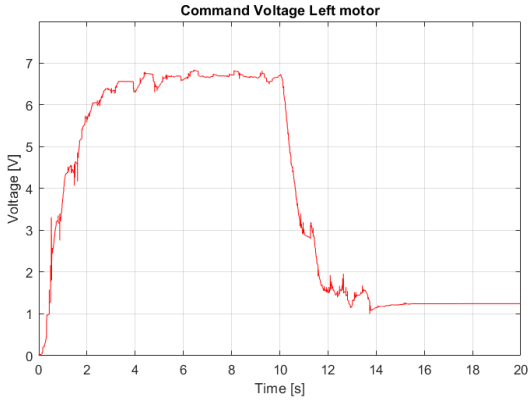


(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

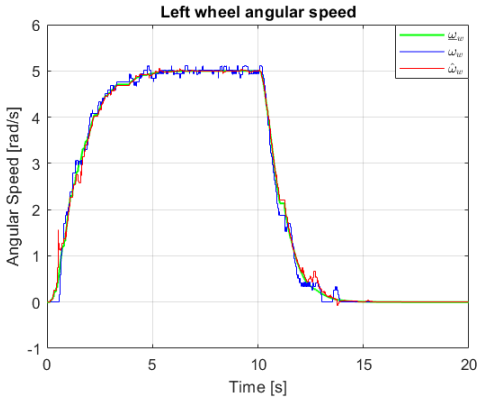


(d) sampling times

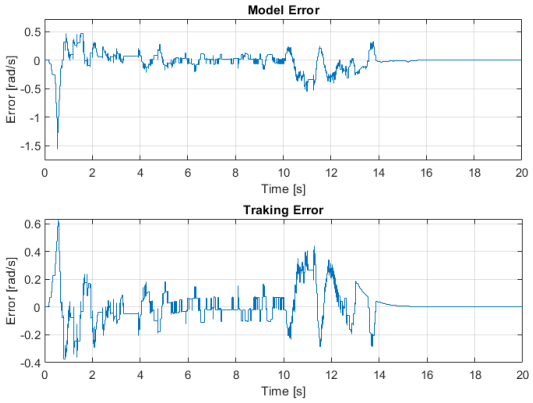
Figure 7.15: right wheel remote control, WiFi connection, test 2 results



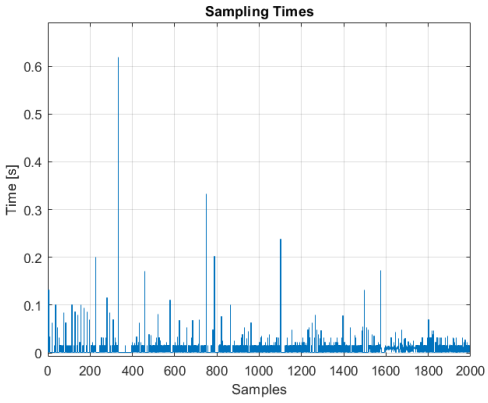
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)



(d) sampling times

Figure 7.16: left wheel remote control, WiFi connection, test 2 results

7.2.6 Results Summary

The first thing to point out is the large difference in scale between the sampling times obtained from the tests on the p2p connection and those obtained from the tests on the WiFi connection. So this is a great example of how remote control system performances change when the variability of sampling times changes as well. Moreover it is worth to notice that, as anticipated, the sampling times have a specific rationality in the realization: as can be seen, for example, in figure 7.16d, very low values are in correspondence of too high peaks and this behavior is attributable to the the fact that an excessive delay of a packet transmission leads to a very fast arrival of subsequent packets. Please note that this event can cause a packet exchange if it happens in more complex Network structures, but can be

certainly excluded for both the Network architecture studied here. In addition, although a UDP protocol is used for the tests performed with WiFi connection, no packets dropouts are registered.
Once made these clarifications, it is possible to better analyze the responses of the controlled systems by comparing the tests performed with the two types of connection. As you can easily observe, the results provided in the last two tests have more deteriorated performances than those of the first two tests, proving that the the Asynchronous-EMC performances are mostly affected by the variability of the sampling times. More specifically, in figure 7.16 it is possible to see that an unlucky realization of the sampling times is obtained, nevertheless it is interesting to see that the wheel speed is kept always very close to the reference. The usual initial oscillation due to the dead-zone detachment can be more evident in some realizations (e.g. in Fig. 7.14b) than in other ones, even applying exactly the same EMC design to the same system.
Another aspect to underline is about the braking phase, which show a control action slightly less precise in the tests conducted with the WiFi connection, since a sort of bounce is experienced when the wheel approaches to the dead-zone, proving, once again, a lower precision in control when the sampling times take on very variable values. However all the results show an excellent behavior of the control system, especially for WiFi connection, which is the one taken as reference since although on a smaller scale, it is still a realistic implementation of a connection between robot and remote server.

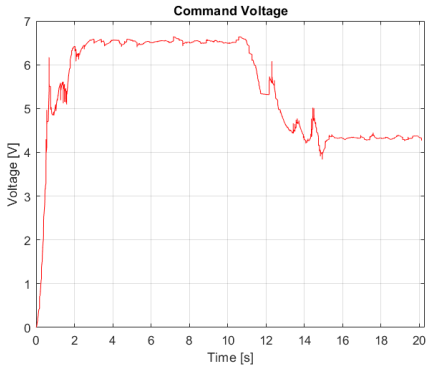
7.3 EMC for Real Motor-Wheel System: Remote Control with Packet Dropout

When it comes to network communication through UDP transport layer protocol, it is necessary to take into account the possibility of information leakages. In previous chapters it was specified that the problem of packet loss would not be addressed in this thesis, since it would require at least a mechanism of fast recovery, to implement on the robot board, which must be activated in case of either a too frequent packet dropouts or a too long waiting time from the last command arrival. Anyway, despite this control design doesn't handle these kind of events, it is very interesting to evaluate its behavior with packet losses homogeneously distributed on the whole transmission. For this reason, it is investigated this aspect by testing the performance of the controller in the presence of a constant rate packages dropouts. In particular six tests will be shown with a rising dropout rate, from now on referenced to as f_d. The latter value indicates the number of lost packets out of the number of sent packets, e.g. if $f_d = 1/5$ then there is one loss exactly after five packet transmissions (it is always the last packet of the sequence to be lost).

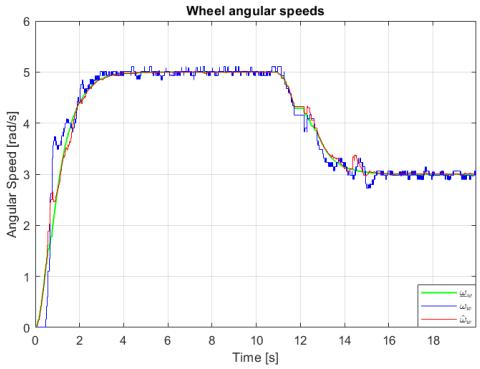
Moreover it is noteworthy to say that packet drop is forced, since having packet loss in a single router network setup is very unlikely, if not impossible, although UDP is used. To simulate the packet drop it is imposed a filter on the server ingress which has the task of discarding packets every $\frac{1}{f_d}$ packets. In addition it must be noted that the following tests are for the left wheel only in order to avoid adding further results about the right wheel which may be redundant information. Finally the reference provided to the wheel for all the following tests is the one represented in 6.1.

7.3.1 Wheel Remote Control With Losses: test 1

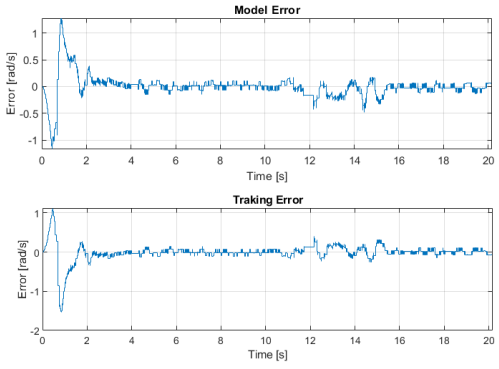
For this test a drop rate of $f_d = \frac{1}{50}$ is used, this means that the number of packets lost in the transmission is the 2% of the total packets. The results are represented in Fig. 7.17.



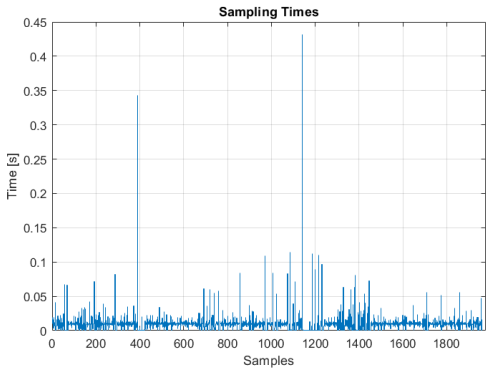
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

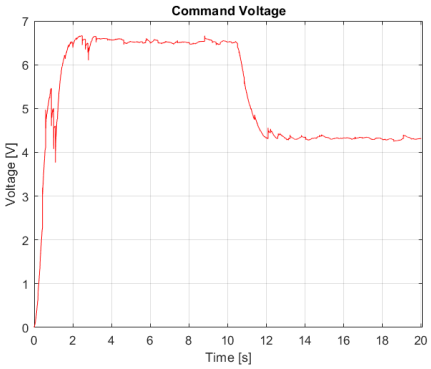


(d) sampling times

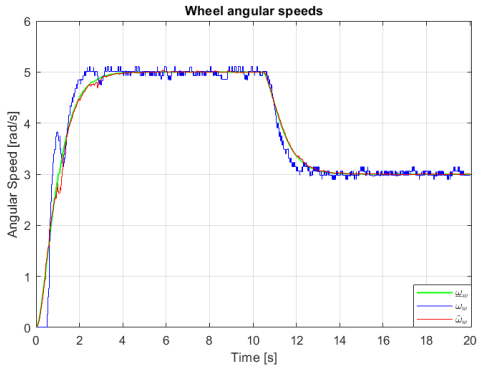
Figure 7.17: remote control on WiFi connection, with 2% packet lost

7.3.2 Wheel Remote Control With Losses: test 2

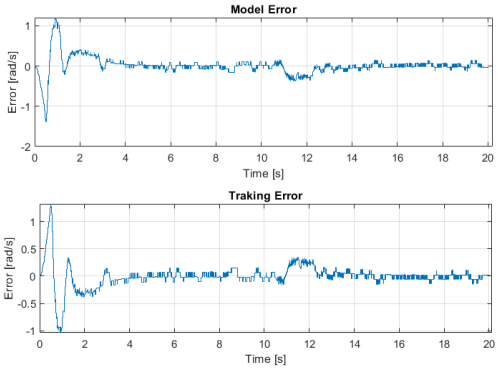
The dropout rate here is $f_d = \frac{1}{20}$, with a percentage of packets lost of the 5% of the total transmitted packets. The results are represented in Fig. 7.18.



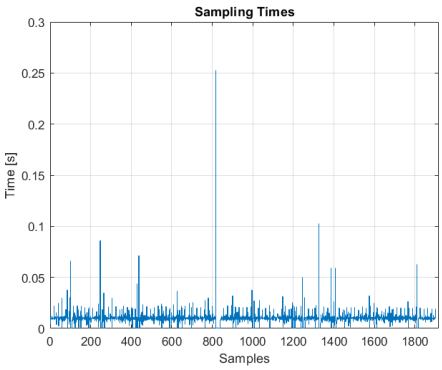
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

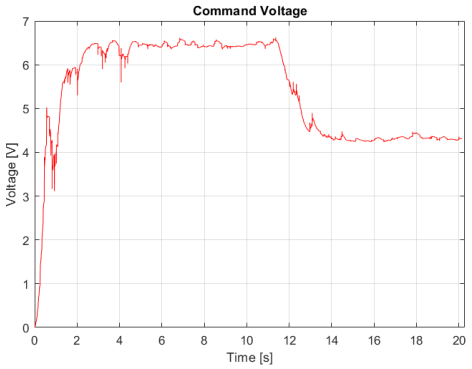


(d) sampling times

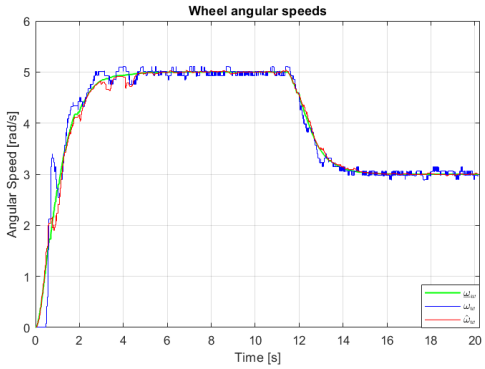
Figure 7.18: remote control on WiFi connection, with 5% packet lost

7.3.3 Wheel Remote Control With Losses: test 3

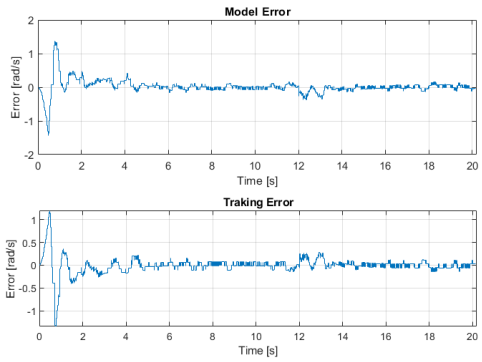
In this test it holds $f_d = \frac{1}{10}$, thus 10% of the total transmitted packets are lost. The results are represented in Fig. 7.19.



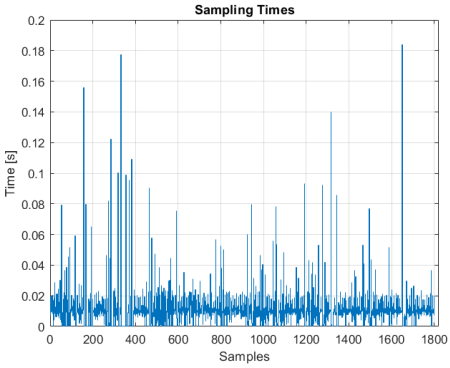
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

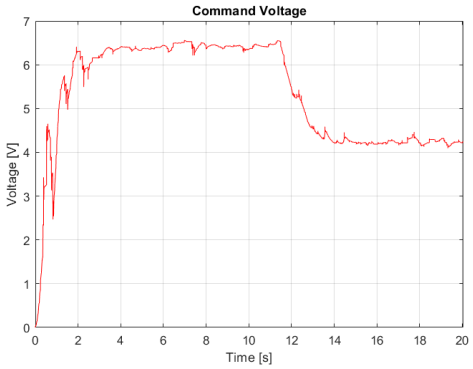


(d) sampling times

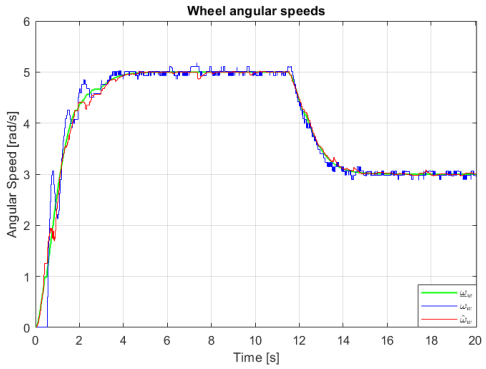
Figure 7.19: remote control on WiFi connection, with 10% packet lost

7.3.4 Wheel Remote Control With Losses: test 4

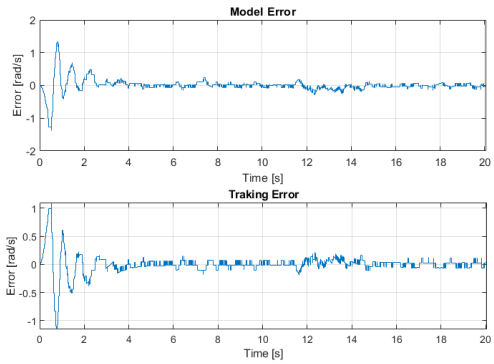
For this test the dropout rate is $f_d = \frac{1}{5}$, that is to say that 20% of the total transmitted packets are lost. The results are represented in Fig. 7.20.



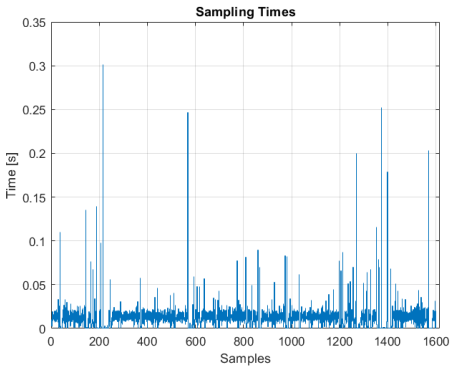
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)

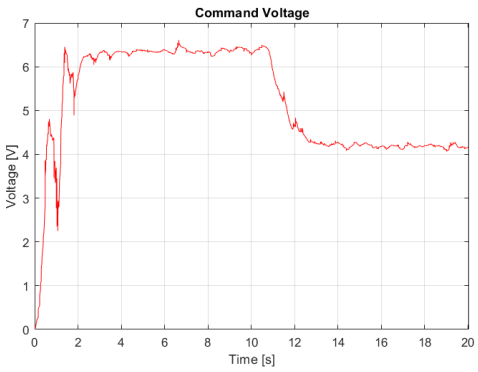


(d) sampling times

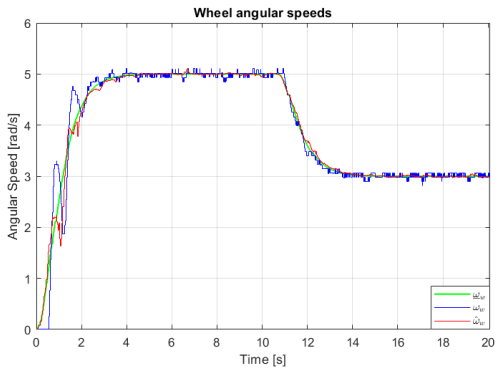
Figure 7.20: remote control on WiFi connection, with 20% packet lost

7.3.5 Wheel Remote Control With Losses: test 5

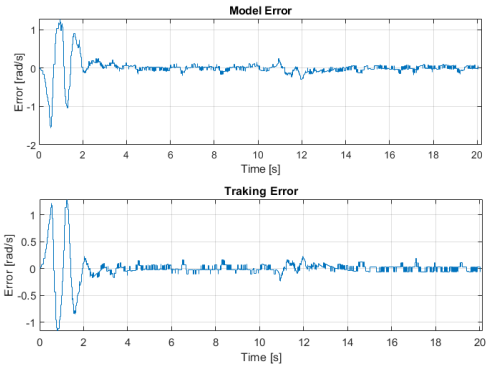
Finally for the last test is used the dropout rate $f_d = \frac{1}{3}$, that means about 33.3% of the total transmitted packets are lost. The results are represented in Fig. 7.21.



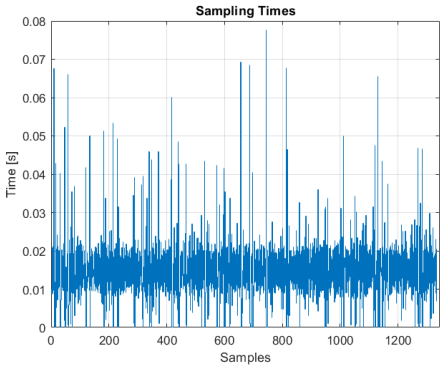
(a) controller command output u



(b) measurements (blue), estimate (red) and reference (green)



(c) model error $\bar{e}$ (top) and tracking error $\underline{e}$ (bottom)



(d) sampling times

Figure 7.21: remote control on WiFi connection, with 33.3% packet lost

7.3.6 Results Summary

The first observation to do is that the designed control seems to respond very well to the information leak, in fact, apart from the usual phase of detachment from the dead-zone, the wheel speed seems to correctly follow the whole reference. Of course, as the dropout rate increases, the control deteriorates, but it is also true that, after a few oscillations, the control settles down quite quickly for all the dropout rates exposed.

Anyway, it is important to underline that this doesn't mean that the asynchronous-EMC is also a solution for the reliability problem introduced by a communication based on the Network in the NCSs, but it is a good proof of the fact that this control system can resist to sporadic information loss, and also to frequent but homogeneously distributed packet leakage. In fact some possible events that can put a strain on this design is the loss of an entire block of packets, in which case it may be appropriate to carry out recoveries specifically designed on the robot.
It is also worth to notice that these tests only concerns the packet loss in the transmission from robot to server, but the loss in the reverse transmission is not taken into account. It can seem an useless experience, but the two behaviors can be totally different, in fact it suffices to think that if the packet drops before entering in the server, the EMC simply skips the measurement, as it was never performed on the plant. On the other hand, if the measurement arrives to the server, but the command resulting from the execution of the EMC doesn't arrive to the robot, then it can also be a less problematic event. In fact let's remind that, even when no dropouts are admitted, it is not mandatory that all the commands are executed on the plant because the actuation is performed every $T_a = 20\,ms$ and this means that if more then one command arrives to the plant in this interval, then only the last arrived is effectively given as input to the motor, while the previous ones are naturally discarded.

7.4 Comparison with PID Controller

An alternative controller that can be applied to an NCS is the PID, [6]. This controller is widely used in many generic applications thanks to the simplicity of its design, and, according to some studies, it can be used also for applications where timing problems are introduced in the control loop. In this section a comparison between the results obtained respectively by PID and EMC controllers is provided, in order to explain why EMC is preferred over PID-based control.
First of all let's start describing the PID design phase. Starting from the transfer function of a PID controller expressed in *Laplace* domain as:

$$C(s) = K_p + \frac{K_i}{s} + K_d \frac{N}{1 + N/s} \tag{7.3}$$

it is possible to obtain its discrete time version by using the forward Euler method, which allows to write the z-domain transfer function as follows:

$$C(z) = K_p + K_i \frac{T}{z-1} + K_d \frac{N\,(z-1)}{z - 1 + N\,T} \tag{7.4}$$

where T is the sampling time.
The tuning of the parameters K_p, K_i, K_d and N is performed by means of a trial

and error procedure over the model of the motor-wheel system described in 5.1, by evaluating the results obtained applying the DT PID to the simulated plant without any kind of noise. At the end of this procedure the resulting values are:

$$K_p = 0.001, \quad K_i = 5, \quad K_d = 0.1, \quad N = 2.$$

Once the parameters are achieved, it is possible to use the related PID for the remote control of the wheel, in particular the WiFi connection is exploited for this application.
In the following some results of the tests performed applying both the PID and the EMC on the remote control of the left wheel are shown.

7.4.1 PID and EMC controllers comparison: test 1

For this test the target provided to both the controllers is $\omega_{ref} = 5\,rad/s$, the figure 7.22 shows the resulting outputs for both the control methodologies, comparing it to the provided target.

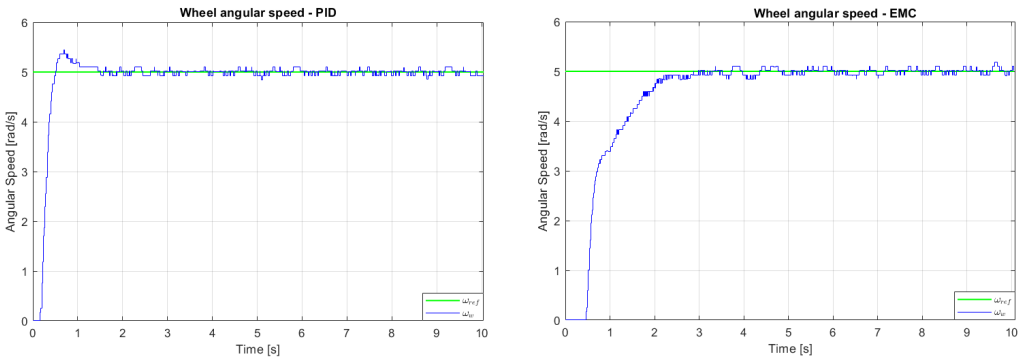


Figure 7.22: PID (left) and EMC (right) remote control comparison, test 1

7.4.2 PID and EMC controllers comparison: test 2

The target $\omega_{ref} = 8\,rad/s$ is provided for this test. Please note that this is an unreachable target since the motors are in saturation when the wheels speed about about $6.5\,rad/s$. The results are on the figure 7.23, showing the total degradation of the PID controller performances.

7.4.3 Results Summary

First of all it is worth to notice that the PID controller behaves very well for this particular application, confirming that it can be a good alternative control

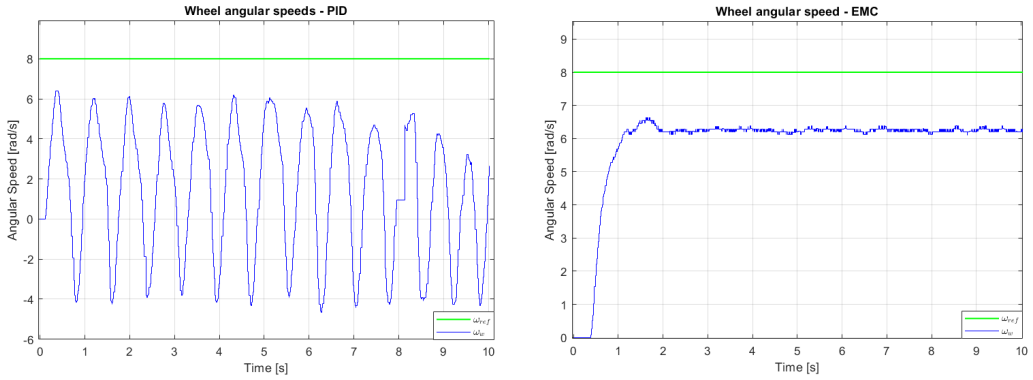


Figure 7.23: PID (left) and EMC (right) remote control comparison, test 2

methodology for NCS. In fact, apart from an initial overshoot shown in the result of tests 1, the tracking of the target is quite accurate despite the variable delays occurring on the measurements transmissions. Since the network constructed for these tests has a small scale, there is not the possibility to show that the EMC presents a better robustness on the timing problems introduced by NCS. Anyway it is expected that for greater sampling time variations the behavior of the PID control degrades, since it hasn't any knowledge of the model, differently from the EMC which is a model-based control methodology, resulting more robust on timing disturbances introduced by the CN.
Anyway some problems can be remarked for PID controller. First of all although PID is quite simple to implement, it doesn't give the possibility to consider constraints on the plant as, for example, the command saturation. This is highlighted form the second test, where a reference which saturates the motors is provided to both the controllers. From this test it can be seen that the PID can't follow the target as it doesn't know that the command saturates. On the other hand the EMC behaves very well and, even if it can't make the wheel reach the target, it makes the wheel settle to its maximum speed. This is due to the fact that the EMC uses the information related to the command saturation included in the reference generator. In addition, another problem of PID could be related to the choice of the response speed, which may not always be easy to impose without making a trade-off with the performances of the control. In fact it can be observed from test 1 that the response is too fast, and it can be a problem especially in this application since a too high acceleration of the wheel can make it slip. With the EMC this problem is solved by giving a smoother dynamic to the reference to track by means of the reference generator.

Chapter 8

Conclusions

Considering the results achieved during the tests, it is possible to confirm the validity of the asynchronous-EMC for remote control applications. It is important to remark that the control system is able, first of all, to guarantee a total rejection of parametric errors of which the model supplied to the EMC is strongly affected (see Figs. 5.6a, 5.6b). This behavior can be seen for both on board and remote controls, showing that the main characteristics of the EMC are not lost if it is executed remotely.
The other great achievement is the robustness of the control to the variability of T_i. This can be seen from both simulations and tests. In fact the simulations shown in section 6.3 are purposely built to verify how the control reacts when the variability of the sampling times increases or when their stochastic realization is particularly unlucky. They show that, although the control loses precision, it still guarantees stability when highly variable sampling times are experienced. This result is confirmed also with the tests of section 7.2, where is shown that, even if the p2p network architecture provides a less variable T_i than ones of the WiFi connection, the control results to be excellent in both the type of connections.

In any case, these results are extremely positive as regards the application of AEMC on networked control systems, nevertheless there are some possible improvements to perform on the EMC design for wheels speed control, proposed in this thesis. In fact, as you can easily guess from the results of the tests proposed in sections 7.1 and 7.2, the most complex obstacle to overcome is the dead-zone of the motor, i.e. the range of input voltage values such that the wheel rotational speed is zero. As widely explained in the previous sections, the dead-zone causes an oscillation both when the wheel starts and when the wheel brakes, since they are two moments in which the input voltage enters the dead-zone range. It should be noted that this is not a cause of instability, but it is a source of inaccuracy in tracking the target. A possible solution for this problem could be to insert this

non-linearity into the reference generator, as done for the command saturation. However, some complications could arise in identifying the range of the dead-zone, since it depends potentially on numerous factors: the motor manufacture, the inertia of the robot, the speed of the wheel and so on. For this reason this solution may not be as simple as it has been for the motors saturation problem.

A final comment should be made on the control system of the entire robot. As the project looks at present, an easy way to control the robot mobility could be to implement an application, running on the server, that offers to the user some basic commands such as "go forward", "go backward", "turn right" and "turn left". Converting these commands to speed references for the wheels and by associating each of them to a key on the keyboard (or of a pad), then a user would be able to maneuver the robot remotely.

A more attractive proposal requires a totally autonomous control system for the robot mobility. Therefore, future additions to this project may concern the implementation of a complete control of the robot, which is able to get as reference a point of the 2D Cartesian plane, making the robot to reach autonomously the indicated point by controlling both its longitudinal position and its orientation, simultaneously. A solution for this type of problem could be the *hierarchical control structure*, for which the position and orientation controls are independent of each other, and the commands generated from their control laws are fictitious force and torque, respectively. These two commands are fictitious because they cannot be directly applied to the system, but they are supplied through the movement of the wheels, which, under the assumption of pure rolling, generate frictional forces that make the robot to translate and to rotate. For this reason the output of both position and orientation controls are the speeds of the wheels which must be supplied as a reference to the controller of the wheels. In this way the force and torque commands are indirectly applied to the robot. The great advantage of the hierarchical control structure is that the individual controllers are extremely simple to implement, since the problem of the actuation of the motors and that of the dynamics of the chassis are separate. The alternative is to study the complete dynamics of the robot through classical analysis systems such as Newton's laws or Lagrange's equations, and then to obtain the dynamic equations that rule the entire mobility of the robot.

Acronyms

NCS
Network Control System

CN
Communication Network

EMC
Embedded Model Control

AEMC
Asynchronous-Embedded Model Control

MPC
Model Predictive Control

PID
Proportional-Integral-Derivative

EM
Embedded Model

CoG
Center of Gravity

LTI
Linear Time Invariant

OS
Operating System

DC

Duty Cycle

DT

Discrete Time

CT

Continuous Time

wrt

with respect to

API

Application Programming Interface

UDP

User Datagram Protocol

TCP

Transmission Control Protocol

PAT

Port Address Translation

Bibliography

[1] Fei-Yue Wang and Derong Liu. In: *Networked control systems: theory and applications*. Springer, 2010, pp. 1–5 (cit. on p. 1).

[2] Carlos Perez-Montenegro, Luigi Colangelo, Jose Pardo, Alessandro Rizzo, and Carlo Novara. «Asynchronous Multi-rate Sampled-data Control: an Embedded Model Control Perspective». In: *2019 IEEE 58th Conference on Decision and Control*. Nice, France, Dec. 2019, pp. 2628–2633 (cit. on pp. 2, 20).

[3] G. P. Liu Y.-B. Zhao and D. Rees. In: *Integrated predictive control and scheduling co-design for networked control systems*. 2008, pp. 1–8 (cit. on p. 3).

[4] Yun-Bo Zhao, Guo-Ping, LiuYu Kang, and Li Yu. In: *Packet-Based Control for Networked Control Systems*. Springer, 2018, pp. 1–15 (cit. on p. 3).

[5] Alberto Bemporad, Maurice Heemels, and Mikael Johansson. In: *Networked Control Systems*. Springer, 2010, pp. 149–152 (cit. on p. 3).

[6] Lasse Eriksson. In: *PID Controller Design and Tuning in Networked Control Systems*. 2008, pp. 1–10 (cit. on pp. 3, 102).

[7] Enrico Canuto. «Embedded Model Control: Outline of the theory». In: *ISA Transactions* 46 (Jan. 2007), pp. 363–377 (cit. on pp. 5, 9).

[8] Fulvio Risso and Loris Degioanni. «An Architecture for High Performance Network Analysis». In: *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001)*. Hammamet, Tunisia, July 2001 (cit. on pp. 29, 30).

[9] *Linux Manual online man7.org*. `https://man7.org/linux/man-pages/` (cit. on p. 35).

[10] G. Malnati, L. Giannantoni, and M. Mosso. slides for Giovanni Malnati's Programmazione di Sistema course. 2018 (cit. on p. 37).

[11] *GitHub repository for GoPiGo3 design informations*. `https://github.com/DexterInd/GoPiGo3` (cit. on p. 41).

[12] *Dexter Industries techincal specifications of GoPiGo3 design.* https://www.dexterindustries.com/GoPiGo/learning/technical-specifications-for-the-gopigo-raspberry-pi-robotics-kit/ (cit. on p. 42).

[13] Luca Nanu, Carlos Perez-Montenegro, and Carlo Novara. «Asynchronous Embedded Model Control For Robotic Applications». Master Thesis. MA thesis. Politecnico di Torino, July 2020 (cit. on pp. 43, 54).

[14] Silbershatz, Galvin, and Gagne. slides for Gianpiero Cabodi's Programmazione di Sistema course. 2018 (cit. on p. 48).

[15] *Linux Manual Reference for Struct spi_ioc_transfer.* https://docs.huihoo.com/doxygen/linux/kernel/3.7/structspi__ioc__transfer.html#ab32597ad72699fd3481059340fdae62c/ (cit. on p. 48).