Double Master's Thesis

Master's Degree in Industrial Engineering (MUEI)

Master's Degree in Automatics and Robotics (MUAR)

Robotic Cloth Manipulation: Real Implementation using Model Predictive Control and Reinforcement Learning

MEMORY

Author: Adrià Luque Acera

Directors: Adrià Colomé Figueras
Carlos Ocampo Martínez

Date: September 2021

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Abstract

In recent years, robots have expanded past factories and industrial environments, being used for urban and assistive applications. Given many of our daily life tasks involve handling non-rigid textile objects, robotic cloth manipulation is a relevant field of study nowadays. For example, a robot can assist a person with reduced mobility in tasks like dressing up, folding clothes or extending a tablecloth. However, highly deformable objects present complex behaviors, adopting multiple configurations during manipulation, making robotic implementations more challenging.

The present Double Master's Thesis explains the process of designing a Model Predictive Controller for cloth manipulation tasks, improving it with Reinforcement Learning techniques to find the optimal model parameters and controller tuning, and finally implementing the full closed-loop control scheme in a real setup, to test its performance.

The developed controller includes an already existing linear model of the cloth, to consider not only the present configuration, but also predict the future behavior of the cloth depending on the chosen control inputs and states. Concretely, it is assumed that some key points of the manipulated cloth piece (corners) must follow a defined trajectory in space without controlling them directly with a robot. This controller was tested in simulation using a more complex nonlinear cloth model, resulting in successful reference tracking almost in real-time.

Both the linear cloth model and the resulting controller presented multiple parameters to be tuned for each specific case. To obtain their optimal values, Reinforcement Learning techniques were applied with successful results, enabling applications in more varied conditions with better tracking performance.

The developed control scheme was then implemented in a real setup, using a WAM robot, an existing Cartesian controller, and adapting previously developed Computer Vision algorithms for cloth identification and segmentation to obtain real feedback data. All these systems were connected using ROS, and adapted to work together in real-time. Several experiments were conducted to add a filtering process and find the optimal working conditions empirically, and then the implementation was tested in more adverse conditions (rotations, fast movements and disturbances), all with positive results and low tracking errors, validating the developed implementation as a suitable one for robotic cloth manipulation tasks with trajectory tracking.

Contents

Abstract **3**

List of Figures **8**

List of Tables **9**

Notation **11**

1 Introduction **15**

1.1 Objectives . . . 15
1.2 Scope . . . 16
1.3 Thesis Outline . . . 17

2 Model Predictive Control for Cloth Manipulation **19**

2.1 Theoretical Background . . . 19
2.2 Problem Definition . . . 25
2.3 Starting Point: Models and Baseline Controller . . . 27
2.4 Controller Redesign . . . 32
2.5 Generalizations and Improvements . . . 38
2.5.1 Accepting Models of any Size . . . 38
2.5.2 Using a Local Cloth Base . . . 41
2.5.3 Outputting a Single TCP Pose . . . 46
2.5.4 Using other Models . . . 48
2.5.5 Minimizing the Slew Rate . . . 53
2.5.6 Simulating Closer to Reality . . . 57
2.5.7 Flexibility Options and Summary . . . 63

3 Reinforcement Learning Enhancements **65**

3.1 Theoretical Background . . . 65
3.2 Learning the Parameters of the Linear Model . . . 70
3.2.1 Gathering and Processing Real Data . . . 71
3.2.2 Learning with Real Cloth Data . . . 74
3.2.3 Analysis of Results . . . 76
3.3 Learning the Parameters of the Controller . . . 82
3.3.1 Initial Considerations . . . 83
3.3.2 Obtaining the Most Suitable Structure . . . 86
3.3.3 Tuning the Resulting Controller . . . 89

ETSEIB

4 Real Implementation **95**
4.1 Overall Structure and Considerations . . . 95
4.2 Adapting the Model Predictive Controller . . . 98
4.2.1 Code Translation to C++ . . . 98
4.2.2 Integration in the ROS Structure . . . 100
4.3 The Cartesian Controller Nodes . . . 102
4.4 The Vision Feedback Nodes . . . 104
4.4.1 Obtaining the Cloth Mesh . . . 105
4.4.2 Robot-Camera Calibration . . . 106
4.4.3 Closing the Loop . . . 109
4.5 Implementation in Real Time . . . 113

5 Experimental Results **117**
5.1 Effects of Working in Real Time . . . 117
5.2 Vision Feedback Filter Selection . . . 120
5.3 Analysis of Control Parameters . . . 123
5.4 Tracking in Adverse Conditions . . . 127

6 Budget and Impact **135**
6.1 Project Budget . . . 135
6.2 Environmental Impact . . . 137

Conclusions **139**

Acknowledgements **143**

Bibliography **145**

ETSEIB

List of Figures

2.1 Basic MPC block diagram in a simulation scenario (full-state feedback) 24
2.2 MPC block diagram. General case with disturbances and state estimation (output feedback) 24
2.3 Two UR10 arms manipulating a cloth piece, picking up two corners 25
2.4 Visualization of the tracking problem 26
2.5 Linear mass-spring-damper model, with $N = 4 \times 4 = 16$ nodes 27
2.6 Closed-loop block diagram of MPC applied to cloth manipulation in simulation 31
2.7 Results obtained executing the baseline MPC implementation 31
2.8 Results obtained executing the Redesigned MPC implementation 38
2.9 Mesh node numbering and connections (4×5 example) 39
2.10 Corner trajectories rotating the cloth trajectory and/or parameters 41
2.11 Evolutions obtained rotating the trajectory and swapping the parameters accordingly . . 42
2.12 Top view of a cloth piece and representation of local parameters and projections 42
2.13 Process to obtain the local cloth base 43
2.14 Worst cases for the two considered methods of computing the cloth base 44
2.15 Results for a new trajectory with rotations, using a local cloth base 45
2.16 New setup with a rigid piece between the TCP and upper corners of the cloth 46
2.17 Comparison between the new nonlinear model and the linear one (1 s movement) 49
2.18 Comparison between the new nonlinear model and the linear one (0.2 s movement) . . . 49
2.19 Comparison between three scenarios with the same maximum speed 50
2.20 Large 13×13 mesh with nodes taken on reduced 7×7 and 4×4 meshes highlighted . . 51
2.21 Unstable evolutions when changing model size or sampling time but not the parameters . 52
2.22 Analysis of the effects of H_p minimizing u with $Q_k = 0.05,\ R = 1$ 55
2.23 Analysis of the effects of H_p minimizing Δu with $Q_k = 0.05,\ R = 1$ 56
2.24 Analysis of the effects of H_p minimizing Δu with $Q_k = 0.2,\ R = 1$ 56
2.25 Resulting distributions for the results of the DMPC and SMPC simulations 59
2.26 Results for a simulation using SMPC 60
2.27 Closed-loop block diagram of MPC applied to cloth manipulation in simulation 61
2.28 3D plot with the results of a simulation, including the WAM Robot 62

3.1 Basic diagram of Reinforcement Learning 65
3.2 3D plots of all the used input TCP trajectories 71
3.3 Raw data gathered through Computer Vision 72
3.4 Comparison between raw and filtered data for the lower left corner 73
3.5 Zoomed in comparison between original data points and regularized ones 73
3.6 Evolution of means and deviations of all parameters in one learning experiment 76
3.7 Evolution of the rewards for the same learning experiment 77

ETSEIB

3.8 Comparison between learnt parameters and their rewards depending on input trajectory . 78
3.9 Rewards for all the considered experiments, grouped by n and T_s 79
3.10 Example of computing the final model parameters for $n = 4,\ T_s = 10$ ms 80
3.11 Simulation results with a longer trajectory in more demanding conditions 82
3.12 Different possibilities to represent two weights with the same proportion 84
3.13 Results of the first round of learning experiments (RMSE) 87
3.14 Results of the first round of learning experiments (time) 87
3.15 Results of the learning experiments using the triple model scheme 89
3.16 Results obtained by simulating across all possible Q, R values 90
3.17 Results obtained by executing 2 epochs of REPS with 100 samples each 91
3.18 Results for all the analyzed trajectories 91
3.19 Situation where a strict upper bound would leave the optimum out (a), and solution (b) . 92
3.20 Comparison of results using the triple model scheme with disturbances 94

4.1 Full diagram of the final implementation in ROS 96
4.2 Picture of the WAM used in the real setup, and piece that connects to the cloth 97
4.3 Picture of the full real setup during an experiment 97
4.4 Results obtained with the translated closed-loop simulation in C++ 99
4.5 Diagram of an open-loop implementation in ROS using the Read Node 104
4.6 A Kinect camera, used to capture RGB-D images and obtain current mesh positions . . . 105
4.7 Diagram of the setup with World, End-Effector and Camera coordinate frames shown . . 106
4.8 Comparison between the different considered filters in a scenario without noise 110
4.9 Diagram of the ROS implementation in real time 113

5.1 Experimental results without Vision feedback nor running in real time 117
5.2 Experimental results with Vision feedback but not running in real time 118
5.3 Experimental results with Vision feedback and running in real time 119
5.4 Evolution of the X-axis position of the lower right corner for all considered filters 120
5.5 Detail of the found situation where the corners of the Vision mesh are not placed accurately 121
5.6 Results obtained on the filter selection experiments 122
5.7 Obtained RMSE using different linear model sizes (5 experiments each) 123
5.8 Results of all the executed experiments depending on $T_s,\ H_p,\ W_V$ 124
5.9 Cloth corners and TCP evolutions for $T_s = 20$ ms, $H_p = 25$, $W_V = 0.2$ 126
5.10 Cloth corners and TCP evolutions tracking a trajectory with rotations 127
5.11 Comparison between evolutions of the same cloth corner executing at different speeds . . 128
5.12 Cloth corners and TCP evolutions blocking the camera in two instants 130
5.13 Human walking in the setup, creating a barrier between camera and cloth 131
5.14 Human agent pulling and pushing the elbow joint during an execution 132
5.15 Disturbance created by a person poking the cloth 132
5.16 Cloth corners and TCP evolutions blocking the camera and with human interaction . . . 133

List of Tables

2.1 Comparison of KPIs between the original and the redesigned controller 37
2.2 Original linear model parameters, tuned using the nonlinear one 41
2.3 Quantitative results of DMPC and SMPC simulations 59

3.1 Final linear model parameters, depending on size and T_s 81
3.2 Optimal Q, R weights, using Δu and no Q_a, for all other combinations 88
3.3 Learnt optimal R/Q ratios for different T_s, H_p 93

4.1 DH parameters of the Barrett WAM 102

5.1 Conditions of the filter selection experiments 122

6.1 Complete project budget 136

Notation

Symbol	**Description**
H_p	Prediction horizon
$J(\cdot)$	Objective function
$N(\mu,\sigma)$	Normal distribution of mean μ and std. deviation σ
$n_r, n_c,\ n$	Number of rows and columns of a mesh, or side size if square
$p,\ p_x, p_y, p_z$	Position, x, y or z-component
$r(t),\ r(k)$	Reference (continuous, discrete)
T_A^B	Transform matrix of base A expressed in B coordinates
T_s	Sampling time, time step
$u(t),\ u(k)$	Control signals/inputs (continuous, discrete)
$v,\ v_x, v_y, v_z$	Velocity, x, y or z-component
$x(t),\ x(k)$	System states (continuous, discrete)
$y(t),\ y(k)$	System outputs (continuous, discrete)
$\Delta u(\cdot)$	Slew rate
$\pi(\theta)$	Policy (depending on some parameters)
$\Vert \square \Vert_n$	n-norm, 2-norm by default if n unspecified
$\Vert \square \Vert_M$	Weighted 2-norm (with diagonal matrix M)
$\underline{\square}$	Minimum value, lower bound
$\overline{\square}$	Maximum value, upper bound
$\hat{\square}$	Estimation

Acronym	**Description**
COM	Control-Oriented Model
CSV	Comma-Separated Values
DH	Denavit-Hartenberg (parameters)
DMPC	Deterministic Model Predictive Control
DoF	Degrees of Freedom
EE	End Effector
FK	Forward Kinematics
GUI	Graphical User Interface
IK	Inverse Kinematics
IPOPT	Interior Point OPTimizer
KL	Kullback-Liebler (divergence indicator)
KPI	Key Performance Indicator
LTI	Linear Time-Invariant
MA	Moving Average
MAE	Mean Absolute Error
MPC	Model Predictive Control
MSE	Mean Squared Error
NaN	Not a Number
QCQP	Quadratically Constrained Quadratic Program
QP	Quadratic Program
REPS	Relative Entropy Policy Search
RGB-D	Red-Green-Blue and Depth (images)
RL	Reinforcement Learning

RMSE	Root Mean Squared Error
RoHS	Reduction of Hazardous Substances
SMPC	Stochastic Model Predictive Control
SOM	Simulation-Oriented Model
TCP	Tool Center Point
WAM	Whole Arm Manipulator
YOLO	You Only Look Once

1. Introduction

This first chapter will introduce the Final Master's Thesis, which is part of Clothilde [1], an ongoing research project of the Perception and Manipulation Group in the Institute of Robotics and Industrial Informatics (*"Institut de Robòtica i Informàtica Industrial"*, or IRI), a joint University Research Institute participated by the Superior Council of Scientific Research (CSIC, *"Consejo Superior de Investigaciones Científicas"*) and the Polytechnic University of Catalonia (UPC, *"Universitat Politècnica de Catalunya"*). The overall focus of this ongoing project, which has been active for years with the collaboration of multiple people, is to develop versatile cloth manipulation techniques for robots, using tools from Machine Learning, Computer Vision, and other related fields, while creating a foundation for cloth manipulation theory, models, datasets, tasks and control methods.

The Clothilde project was motivated by the expansion of Robotics into assistive applications, concretely the many daily life tasks involving textile objects, which, due to their deformable nature, present more complex behaviors than rigid objects, making robotic cloth manipulation a challenging field of study. It is also relevant and active nowadays, given the variety of cases it can be applied to. For example, robots could help dress people with reduced mobility on their own, increasing their independence, or wash, fold and put clothes in the closet.

Even though there has been progress and successful results, trying to apply learning algorithms has proven challenging due to how sensible any task is to all the possible disturbances and small variances in the initial conditions, resulting in noise that makes the first step of parametrization into a reward function already quite difficult. One way to solve this problem is with the assistance of predictive control methods, specifically those using efficient cloth models such as Model Predictive Control (MPC). This is the motivation behind the original project proposed by the IRI [2]. After some developments, the present Thesis is centered around the implementation of a specific MPC using existing cloth models into a real robot arm, with the inclusion of real-time feedback, to then improve the task execution performance with Reinforcement Learning (RL) techniques.

Given this context, Section 1.1 describes the intended objectives of the Thesis, Section 1.2 describes and limits its scope, and Section 1.3 briefly summarizes the structure and contents of this document.

1.1 Objectives

As briefly described, the Perception and Manipulation Group at the IRI has been working on cloth manipulation for years, and this project is the result of previous research. Right before the start of this Thesis, some cloth models were developed, and there were even some tests using MPC in simulation. However, this MPC already had some limitations in simulation, was not implemented in a real robot, and also, by its nature, presented a variety of parameters, cost functions and constraints to be tuned.

These shortcomings gave reason to continue this line of work, and can be reinterpreted as the main objectives of the next phase of the research project, i.e., the work described in this Thesis document. Therefore, these objectives are:

- Improving the existing baseline MPC in simulation, generalizing it to more cases and models, and preparing it for real scenarios.
- Implementing a Model Predictive Controller into a real robot, translating and adapting the code as needed. This includes the design of a full closed loop, adding a Cartesian controller and Vision feedback, and their execution in real time.
- Adding Learning techniques to automatically tune all the parameters present in the controller so the tasks are executed optimally, improving the results of the predictive controller. This can be separated in two categories: learning the parameters of the cloth model, and learning the parameters of the controller itself.

1.2 Scope

The original project proposed by the IRI had some progress and was later adapted into "Reinforcement Learning and Visual Servoing for Model Predictive Control", proposed on the IRI-UPC Internship Program 2021-1 under the *"María de Maeztu* Unit of Excellence" Seal, tied to a Research Initiation (INIREC, *"INIciació a la RECerca"* in Catalan) grant at the UPC. This Thesis starts at this point, as a 4-month granted internship at the IRI to continue working on the project, and before starting with the main body of the document, it is worth mentioning its limits and general scope.

First of all, the starting point is not a blank slate. The work carried out at the IRI before starting this Thesis resulted in the creation of nonlinear and a linear cloth models, and also a baseline MPC in a closed-loop simulation. This conforms the foundation of the work done in this Thesis, as will be detailed in Section 2.3.

The fact that the project was proposed from the IRI and had some work done beforehand meant that, from the start, there was not absolute freedom, so the type of control was already decided and set as MPC, and the cloth models to be used were the ones developed, but also meant there was no need to search for other options in the vast research publications. This Thesis covers a project with clear guidelines and builds upon previous work, without considering radically different options, which are to be considered out of scope. This Thesis does not try to create or come up with the best possible control technique for a cloth manipulation application, nor does it involve modeling cloth pieces.

The main scope can be derived from the objectives and the context given in the previous paragraphs. It includes improving and contributing to the previous work, done in simulation, transforming the result of that process so it can be executed in a real robot, connecting all the necessary systems, gathering data and analyzing the results, and using Reinforcement Learning techniques to improve the results even further.

Of course, to create a working control scheme with a real robot, there is a lot of assembly work putting pieces together, and some of them had been developed for other projects at the IRI. This is the case of the code responsible of obtaining a cloth mesh from the data of a camera, detailed in Section 4.4, and the used Cartesian Controller, adapted from an available code as explained in Section 4.3. Once again, the scope of the Thesis does not include developing these codes, but it does include their understanding, adaptation, improvement, usage and connections with the rest of the control scheme.

Finally, it is also worth mentioning that the results of this project (this Thesis plus previous work on the topic done at the IRI, that is out of the scope of this Thesis but forms its basis), will be published on a journal paper. Even though the scope of the paper is a bit larger, this proves the contribution of this Thesis in the general Robotics research.

1.3 Thesis Outline

This document is organized revolving around the three axes described by the three main objectives mentioned in Section 1.1: MPC, RL and the implementation of the complete control scheme into a real robot. Although the table of Contents shows every subsection and links to it, here a small summary of every chapter is presented to help the reader know what to expect and decide what to read.

- **Chapter 2: Model Predictive Control** begins with a theoretical background on control systems and this type of controllers, to then describe the problem at hand, what was done before this Thesis, and the work done involving MPC, starting with a controller redesign, and then a series of minor improvements organized in subsections.
- **Chapter 3: Reinforcement Learning** also has a brief theoretical base at the start, followed by the application of RL techniques in two different ways: first, to improve the modeling of a real cloth piece, learning the parameters of its linear model. Second, to tune the controller automatically, considering all the different options available.
- **Chapter 4: Real Implementation** is dedicated to all the steps taken to go from simulation to a real robot, starting by translating the code. The final system uses ROS (Robot Operating System), so all the nodes and overall structure will be described in this section. Finally, the real system must also work in real time, which implies a series of changes and considerations explained at the end of this chapter.
- **Chapter 5: Experimental Results** shows the outcome of the real implementation, and contains some analyses of the obtained results discussing, for example, types of filters for the Vision feedback or combinations of control parameters.
- **Chapter 6: Budget and Impact** discusses the budget of this project and its effects on the environment, both positive and negative, derived from its applications and resources used.
- **Conclusions** closes the main body of the document reflecting on the work done in relation with the objectives, and describes some future work that can be done in upcoming projects.

2. Model Predictive Control for Cloth Manipulation

This chapter revolves around Model Predictive Control (MPC) and its use in this Thesis. Section 2.1 introduces the necessary theoretical concepts and properties of MPC, while Section 2.2 defines the specific control problem in hand. Section 2.3 specifies the controller developed prior to this Thesis. Section 2.4 introduces its redesign, and then Section 2.5 delves into all the modifications introduced to create a generalized and improved version, ready to be applied to a real scenario.

2.1 Theoretical Background

Model Predictive Control stems from Optimal Control [3]. Its main idea is to use a dynamic model to forecast, or "predict", the behavior of the system, formulating an optimization problem to obtain the control signals that will produce the best outcome. At any given time, one can predict until a determined moment in the future, or horizon, optimize to obtain a sequence of states and control signals, and apply the first one to the system so it evolves into a new state, from which the process can start again.

Dynamic models are usually described by the differential equations on (2.1), where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the control signal or input, $y \in \mathbb{R}^p$ is the output and $t \in \mathbb{R}$ is time. Equation (2.1c) is the initial condition, which specifies the value of the state at the starting instant (usually defined as $t_0 = 0$).

$$\dot{x}(t) = \frac{dx(t)}{dt} = f(x, u, t) \tag{2.1a}$$

$$y(t) = h(x, u, t) \tag{2.1b}$$

$$x(t_0) = x_0 \tag{2.1c}$$

The previous formulation is the most general one, valid for a generic system. A specific but common case is to have the dynamic model as a Linear and Time-Invariant (LTI) one. In that case, the formulation becomes the one shown in (2.2). Even though the time invariance means the matrices do not depend on time, the other three variables are still time functions ($x(t),\ u(t),\ y(t)$). For simplicity, this dependency is commonly not explicitly written.

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \\ x(0) &= x_0 \end{aligned} \tag{2.2}$$

This formulation is also known as the State-Space (SS) Representation, and can also be used for discrete-time models, when the systems are sampled at a given rate. This rate (time step, sampling time

or sampling period, T_s) must be chosen carefully according to the dynamics of the system, and if done properly, the resulting system equations can be seen in (2.3), where instead of a derivative, we obtain the next state sample with data of the current sample, and instead of a differential equation, we have a linear difference equation. Here the corresponding time step must be specified to avoid confusion, either between brackets or as a subscript, depending on the used notation.

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \\ x(0) &= x_0 \end{aligned} \tag{2.3}$$

The previous expressions describe deterministic models, where all the dynamics are described with a State-Space realization, but when studying a real system, we use instruments with a certain precision and uncertainty, and sensors that can have some noise, resulting in fluctuations in the obtained data. Stochastic models try to consider these uncertainties, and their research is an active and important topic nowadays, as improving the techniques to model real complex systems helps designing and testing controllers, and is essential in those that depend directly on the model of the system, such as MPC. The simplest stochastic model is a variation of the previously presented one, but adding a random sensor noise $n(k) \in \mathbb{R}^p$ to the output, and some unmodeled disturbance $d(k) \in \mathbb{R}^g$ that affects the state of the system, as seen in (2.4). Matrix $B_d \in \mathbb{R}^{n\times g}$ is used to apply the effects of the disturbances to the states when the relations are known. Sometimes, each disturbance affects one state (thus $g = n$), so this matrix becomes the identity.

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) + B_d d(k) \\ y(k) &= Cx(k) + Du(k) + n(k) \\ x(0) &= x_0 \end{aligned} \tag{2.4}$$

Now that we know how to represent the model of a system, the main objective is to obtain a way to control it so that the system evolves in a desired way, e.g., as close as possible to a given reference, with the minimum possible error, or optimizing a series of other performance indicators. This means obtaining a sequence of control signals u such that the resulting evolution of the system is as close to the desired one as possible.

Depending on the approach we choose, we obtain different types of controllers. One of the most common ones is using State Feedback, where the control signal directly depends on the current state vector through a gain matrix ($u = Kx$), which is obtained with the given specifications. While the equations presented until now can be defined as open-loop, because the states and outputs of the system depend on themselves and arbitrary independent inputs, feeding the states back to obtain the control signals like this creates a closed-loop system, as we can think of the dependencies between variables as a complete circle or back and forth between states and controls. Although not as simple as a constant matrix, most controllers are based on feeding back output data from the system through different processes.

For this project, a Model Predictive Controller was chosen as the specific option to be tested. Previous research at the IRI had already tested simpler control techniques, which were too sensitive to the initial conditions and possible disturbances of the environment [4], and a first approach to MPC had been tested with satisfactory results, so the choice of MPC itself and the comparative analysis with other control techniques had been done prior to this Thesis and thus are out of its scope. Regardless of that, before continuing, we can list some of the properties of MPC, and its advantages and disadvantages [5], [6], [7], to give a brief background on the basis of the choice. They are:

Advantages:

- Usage of all system dynamics described by the model.
- Multiple and varied control goals can be considered.
- Simple and optimal control policies even on complex systems.
- Straightforward consideration of physical constraints.
- Use of feed-forward features that reject disturbances.

Disadvantages:

- An accurate model of the real system is required.
- Complex models can lead to high computational loads.
- Need of numerical optimization solvers.
- High number of parameters, which must be tuned.

The fact of using a model already comes with its pros and cons, as it gives a lot of information that can be used to predict how the system will evolve and what is the best course of action, but the model must be accurate to reality to have a direct correspondence with the evolution of the real system, and complex dynamics come at the cost of computational load and time. The most important advantage nowadays, however, is probably being able to ask for multiple control objectives, and weigh them as needed (e.g., tracking a reference trajectory while also keeping the energy consumption as low as possible) to set priorities among them.

For the drawbacks, nowadays we can use a wide array of optimization solvers that keep getting more common and less resource intensive as time goes on, and there are techniques to reduce complexity without losing important dynamics on the model, so the main point that always stands is the large number of parameters that need to be tuned. This is the principal reason for the addition of a Learning algorithm that tries to obtain the optimal combination of parameters automatically once the control structure is set and a large number of simulations can be executed.

With the controller type chosen and its overall characteristics described, we can proceed into a formal formulation of a general MPC. The control signals, which are what we are looking to obtain in any controller, are found via an optimization problem. In it, we define an objective function to optimize (usually minimize) and a set of constraints. The basic objective function of MPC measures the deviation of both states and control signals from zero, and takes the form of weighted squares as seen in (2.5), where matrices Q and R weigh the state and control input deviations, respectively, and H_p is the prediction horizon, the number of steps that we look into the future to predict the behavior of the system.

$$J = \sum_{k=1}^{H_p} \left[x(k)^T \; Q \, x(k) \right] + \sum_{k=0}^{H_p-1} \left[u(k)^T \; R \, u(k) \right] \tag{2.5}$$

It is worthy to point out how from the two distinct parts, the state sum goes from 1 to the horizon H_p itself, while the control signals are shifted a step before that. Of course, this is because the states at step $k = H_p$, which is the furthest in the future we want to predict, are obtained with states and control signals at step $k = H_p - 1$, as per (2.3), so the next control signal is not needed to reach the horizon. Additionally, $x(0)$ is the initial condition, so there is no need to penalize it. The last predicted state, at the horizon, is sometimes separated from the rest with a unique weighting matrix to penalize the final deviation even further.

The objective function in (2.5) assumes that the states need to tend to the origin and penalizes (the function will be minimized, so a higher value is worse) their deviation from zero, and the same happens with the control signals. In a general scenario, this is not what we want, as the states will probably need to reach a certain value or to follow a series of values. This is why the most common version of the MPC objective function includes the error, or difference between the states and their reference or intended value, at each time step, as shown in (2.6).

$$J = \sum_{k=1}^{H_p} \left[\big(x(k) - r(k)\big)^T \;\; Q \;\; \big(x(k) - r(k)\big) \right] + \sum_{k=0}^{H_p-1} \left[u(k)^T \; R \, u(k) \right] \tag{2.6}$$

There are, more alternatives, including ones for the control signals. For example, instead of trying to reduce their absolute value to decrease the energy consumption, one might want to reduce the variation between consecutive steps, to reduce how nervous the input is and decrease potential damage in real mechanisms. This can be done by using the slew rate ($\Delta u(k) = u(k) - u(k-1)$) in the objective function instead of the control signal (u) itself. This can be especially useful when disturbances and noise increase nervousness of the resulting control signals.

Besides the objective function, the set of constraints can also be varied. However, it should always include the dynamic equations of the model, as each state and control signal on every time step is a different optimization variable, but they are related in a known way from one step to the next along the horizon. The initial condition must be present too. Other constraints usually include physical limitations

of variables (upper and lower bounds, or rate limits), as in the real world actuators, sensors and systems cannot take any real value or change instantaneously, as well as other hard or soft limits that must be considered [8]. In the end, the general MPC-related optimization problem results in what can be seen on (2.7), where the model equations are expressed as general discrete functions which may be nonlinear, and the new $\mathbb{X}$ and $\mathbb{U}$ are the subsets of their respective spaces that satisfy all bounds affecting states and control signals, respectively. All constraints must hold for the entirety of the prediction horizon.

$$
\begin{aligned}
&\min_{u(k)} \; J(x,u) \\
\text{s.t.}\quad & \left.\begin{aligned}
x(k+1) &= f(x,u,k) \\
y(k) &= h(x,u,k) \\
x(0) &= x_0 \\
x(k) &\in \mathbb{X} \subseteq \mathbb{R}^n \\
u(k) &\in \mathbb{U} \subseteq \mathbb{R}^m
\end{aligned}\right\} \quad \forall k \in [0, H_p - 1]
\end{aligned}
\tag{2.7}
$$

To implement a Model Predictive Controller, the optimization problem in (2.7) must be solved at every time step, updating the initial condition to the current state, predicting an horizon, and then applying only the first control input of the obtained optimal sequence. This means that the index k is a local variable, which always starts at 0 (meaning "current time") and ends at the horizon H_p, which globally would be the moment equal to current time plus this many steps.

In a real application, the controller has a model of the system inside for the optimization problem as described, and its output, the control signal, is fed into the real system, from where we measure some outputs to apply as feedback and close the loop. However, before implementing a controller in the real world, it is common to test it in simulation. Simulating a real system requires an additional model, called Simulation-Oriented Model (SOM), separate from the one used inside the controller, which is the Control-Oriented Model (COM). In the most basic case, the used SOM can be the same model as the COM, so they will evolve equally and the prediction will be perfect, which is not very realistic. An easy way to fix this is adding random disturbances and noise, which will be present in a real case and produce some variation between models. Other ways involve having different models for each case. It is not unusual to simplify the COM as much as possible, without losing important dynamics, with the objective of making the optimization problem easier and faster to solve, so the SOM can be a more complex model of the system, e.g., making the COM a linear model (Linear MPC is much faster and simpler) and the SOM a nonlinear one.

Another issue we can encounter when applying an MPC into a real system is how to obtain the outputs of the system to feed back into the controller. In other words, at every time step the optimization problem needs an initial state to predict from, but in real systems we usually do not have access to all the states, or measuring them could be expensive, slow, or hard to the point of altering the behavior of the system to obtain their value. In these cases, we need to analyze which are the outputs of the system that can be

measured with reasonable ease, and try to estimate the necessary states from them. There exist several options to choose from [9], [10]. To list some of the widely used ones, we have Luenberger Observers, Kalman Filters and their Extended variants (KF, EKF), and Moving Horizon Estimators (MHE). The last ones are based on the same principles as MPC [11], [12], applying an optimization problem on a sliding window with a model of the system, but instead of looking into the future to obtain the optimal control signal to apply, we look into the past known data to estimate the current states.

All in all, the minimal Model Predictive Control scheme in a simulation scenario can be represented as the block diagram seen in Fig. 2.1, where the output of the SOM is the full state vector that can get fed back directly into the MPC block, together with a reference. Inside the MPC, we find the Optimizer and the COM used to obtain initial states and constraints. We can also represent a more realistic scenario, with all the details described in this section, as done in Fig. 2.2, where we have a "Plant" block, which could be the real system or a SOM, which is affected by some disturbance d and outputs a vector y that is not the state vector, so it must go through an estimator before closing the loop with the estimated states $\hat{x}$. The output also receives some sensor noise n in the measurement process.

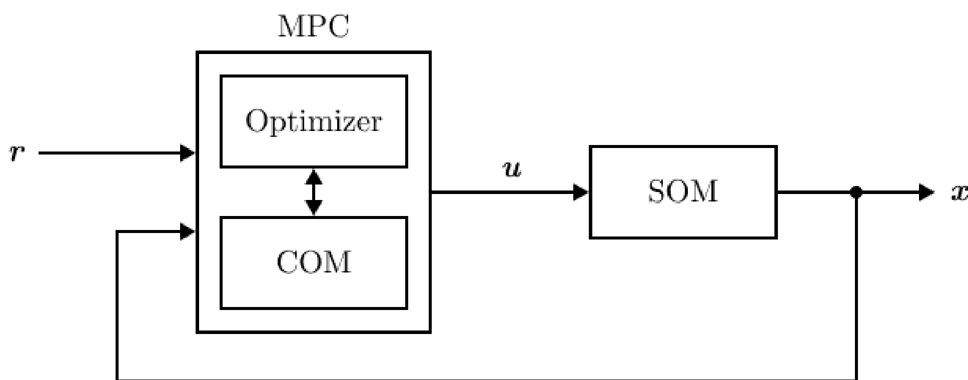

Figure 2.1: Basic MPC block diagram in a simulation scenario (full-state feedback)

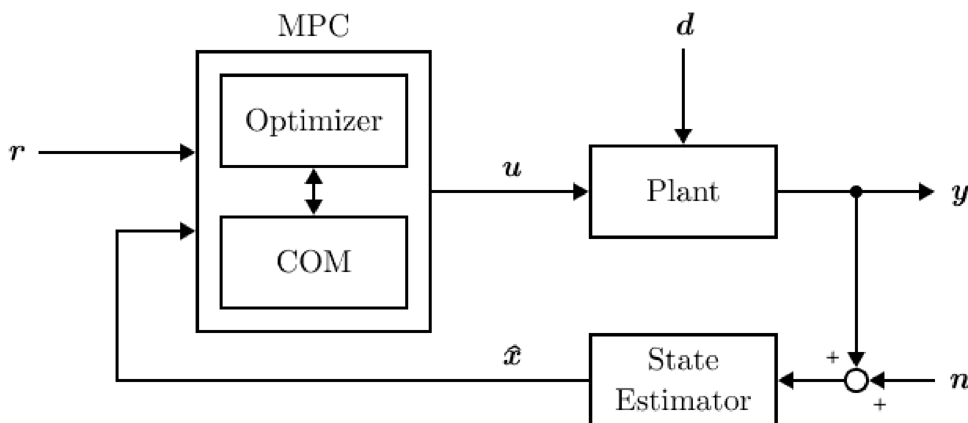

Figure 2.2: MPC block diagram. General case with disturbances and state estimation (output feedback)

2.2 Problem Definition

With the concepts explained in the theoretical introduction to MPC, we can now proceed to explain the specific problem at hand. The overarching objective is to manipulate cloth pieces as accurately as possible with robots, considering they are not rigid objects and they are picked from specific spots, and not taken fully inside a gripper, folded, wrapped or otherwise, as seen in Fig. 2.3. The resulting control objective is simply described as tracking: following a trajectory with the lowest possible error, but the nature of cloth pieces and the environments where they must be manipulated (usually with humans in the very near proximity) increase the difficulty of this task significantly.

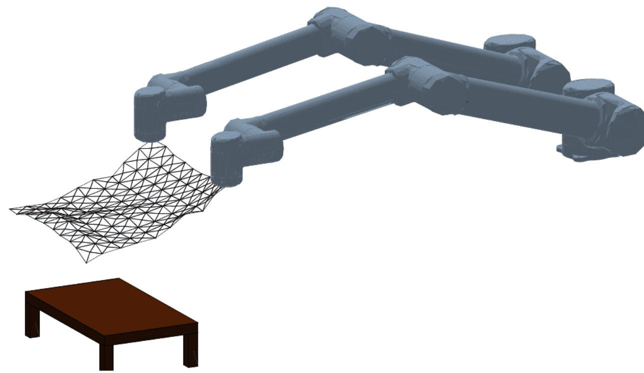

Figure 2.3: Two UR10 arms manipulating a cloth piece, picking up two corners

We could consider both robot and cloth into a big, complex model to use in our predictive controller, where we would have to describe all the dynamics of the robot, consider what needs to be simplified to reduce computation time, decide between a dynamic or a purely kinematic model, and end up with a high order model with joint torques as inputs and cloth positions as outputs, paralleling the real system. The connection would then be direct: the outputs of the controller would be these torques sent to the robot, the feedback signal (sensed, for example, with a camera) would be the cloth positions, and the reference is a trajectory to be followed. This apparent simplicity in the connections, however, does not compensate for the clear complexity inside the controller and its model, having to work with two very distinct subsystems, robot and cloth, with equations describing all the important dynamics well enough to have useful predictions, while being simple enough to compute them fast.

This is one of the main reasons not to follow this approach, and, conversely, separate the robot and the cloth as two systems, focusing only on the cloth piece as the one to be modeled and controlled using MPC. Another reason is that considering the robot as its own system also comes with clear advantages. Now the problem is simply to track a reference trajectory of the End Effector (EE) or Tool Center Point (TCP) of the manipulator, given in Cartesian space (position and orientation in the regular 3-dimensional space). This task is done by Cartesian controllers, a well-known and researched area of robotics that extends beyond cloth manipulation, as they can be used in all kinds of applications. As this controller will not be designed using MPC, it will only be discussed when used for the real implementation of the scheme, in Section 4.3.

We are, thus, left with the cloth piece as our plant to be modeled, with the objective of tracking a defined reference trajectory by controlling (picking up with a robot and moving) only a few points. More specifically, for the implementation developed in this Thesis, a square-shaped piece of cloth will be our plant to be modeled for the MPC. The upper corners will be the controlled points, where the robot will pick the cloth up, and the reference trajectory will be that of the lower corners, as they are the points furthest away from the controlled ones, and because in the majority of applications, there is no need to control how all points in the cloth move. In short, given the cloth model and a reference trajectory of the lower corners in x-y-z-coordinates (r), the controller must find the trajectory of the upper corners (u) that results in the best reference tracking, as we can see in Fig. 2.4.

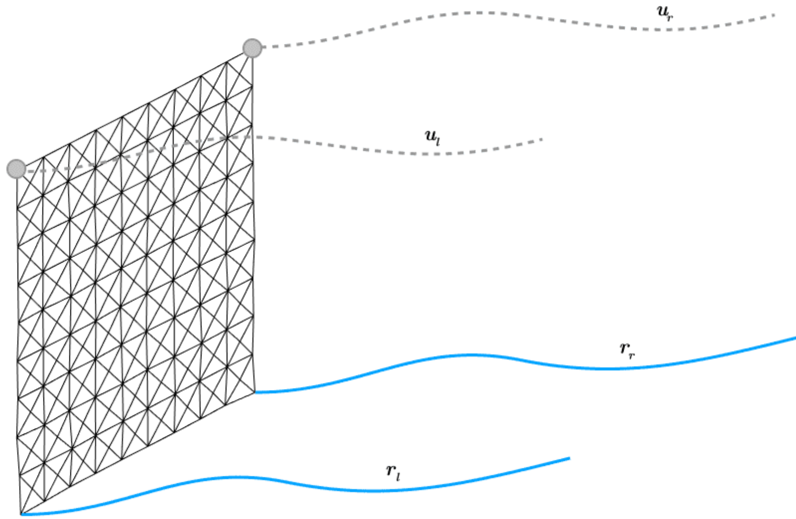

Figure 2.4: Visualization of the tracking problem

More formally, we want to find a sequence of control signals $u \in \mathbb{U} \subseteq \mathbb{R}^6$ ($\mathbb{U}$ being the subset that satisfies the bounds of u, $\mathbb{U} = \{u \in \mathbb{R}^6 : \underline{u} \leq u \leq \overline{u}\}$) such that the left and right lower corners follow a reference trajectory $r = \{r_l, r_r\}^T \in \mathbb{R}^6$ as accurately as possible, i.e., minimizing the error between the resulting real trajectory of the lower corners and the reference.

Controlling the position of the lower corners while only being able to move the upper ones can prove to be a quite challenging task. They are related by nonlinear dynamics, involving both position and velocity of the whole cloth. The usual way of representing a non-rigid object such as a cloth is by transforming it into a mesh of points, such as the one in Fig. 2.4, and describing the dynamics between them with sets of equations. The order of the model increases when the mesh is finer, and this is just an example of the issues found when modeling this type of materials [13]. This is why this task also has its own line of research, and the process of modeling is left out of the scope of this Thesis. Luckily, at the IRI, there has been some recent progress in this topic, and, as discussed in the following Section, 2.3, we have a couple of validated models ready to be used with MPC.

2.3 Starting Point: Models and Baseline Controller

As explained in the Introduction, this Thesis is not an isolated work, but part of a larger project at the IRI, meaning it starts with work done beforehand, and builds and expands upon it. Specifically, it benefits from recent research of Franco Coltraro and David Parent, taking a validated nonlinear cloth model from the former [14], [15] and a linear one plus an initial MPC scheme from the latter [16].

The decision to use their models instead of others found in literature is not purely to follow the research done in the same institution. Most models shown in other papers are focused in describing the complex cloth dynamics accurately, and result in high order nonlinear models, slow to simulate and not well suited for real-time control [17]. Even in recent research regarding cloth manipulation using MPC, simulations take several minutes to finish [18], or the cloth model is left out of the controller [19]. This situation is what led to the creation of custom models, designed to strike a balance between accuracy in representing the cloth dynamics and simplicity to be simulated in real applications.

The nonlinear model can be used as a Simulation-Oriented Model (SOM) in simulation, as it has been validated against data gathered from a real piece of cloth, and serves as its substitute before implementing the control scheme into the real scenario. It follows the equations shown in (2.8), where $\varphi(t) \in \mathbb{R}^{3N}$ is the position of the N nodes of the mesh, ρ is the density, M is the augmented mass matrix, F_g is the force of gravity, K is the stiffness matrix, with θ being a stiffness parameter, D is the Rayleigh damping, with α and β being damping parameters, λ are Lagrange multipliers ensuring inextensibility and C represents inextensibility constraints. Parameters α, β and θ were fitted using real data.

$$\begin{cases} \rho M \ddot{\varphi} = F_g - \theta K \varphi - D \dot{\varphi} - \nabla C(\varphi)^T \lambda \\ C(\varphi) = 0 \\ D = \alpha M + \beta K \end{cases} \tag{2.8}$$

This nonlinear model is quite fast, but is still nonlinear and complex. This is why an even simpler model was developed at the IRI with the purpose of being the Control-Oriented Model (COM) used in the controller, both in simulations and in the real implementation. It is a linear model, based on 3-dimensional mass-spring-damper connections between nodes, considering both their positions and velocities as the states, as represented in Fig. 2.5.

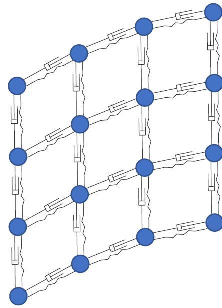

Figure 2.5: Linear mass-spring-damper model, with $N = 4 \times 4 = 16$ nodes

This simplification assumes a loss of some dynamics, but is designed to keep the principal ones present, to be able to predict the evolution of the system correctly and find the optimal control input. With it being a linear model, we can write it in State-Space, as shown in (2.9).

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) + f_{ct} \\ y(k) &= Cx(k) \\ x(0) &= x_0 \end{aligned} \tag{2.9}$$

Here, if N is the number of nodes, $x \in \mathbb{X} \subseteq \mathbb{R}^{6N}$ is the state vector, which includes position and velocity in all three space coordinates (x, y, z). To avoid confusion between the whole state vector $x(k)$ and the position in the x-axis, we will use the variable p to designate positions, and p_x for the x coordinate. Same reasoning applies to the output vector $y(k)$ and the y-axis coordinate. For velocities, we will use the v variable, with a subscript indicating the corresponding coordinate. This means the state vector can be expressed as $x = [p_x, p_y, p_z, v_x, v_y, v_z]^T$, with each of the six parts being an N-dimensional vector. Therefore, $\mathbb{X} = \{x \in \mathbb{R}^{6N} : \underline{x} \leq x \leq \overline{x}\} = \{x \in \mathbb{R}^{6N} : \underline{p_x} \leq p_x \leq \overline{p_x},\ \underline{p_y} \leq p_y \leq \overline{p_y},\ \underline{p_z} \leq p_z \leq \overline{p_z},\ \underline{v_x} \leq v_x \leq \overline{v_x},\ \underline{v_y} \leq v_y \leq \overline{v_y},\ \underline{v_z} \leq v_z \leq \overline{v_z}\}$. The lower and upper bars on variables represent minimum and maximum bounds, respectively. We also have $u \in \mathbb{U} \subseteq \mathbb{R}^6$ as the input vector (position of the two upper corners) and $y \in \mathbb{Y} \subseteq \mathbb{R}^6$, which is the output vector of lower corner positions. Depending on the case, we can also take the full state vector ($y = x$) or the first half (positions, $y = p$), as all positions can be measured even in a real application with some vision feedback.

The full process and theoretical background to obtain system matrices $A \in \mathbb{R}^{6N\times6N}$, $B \in \mathbb{R}^{6N\times6}$, $C \in \mathbb{R}^{6\times6N}$, and the constant force vector $f_{ct} \in \mathbb{F} \subseteq \mathbb{R}^{6N}$ will not be reported here, but a summarized version will be shown for a better understanding of the model used in this Thesis. The basic dynamics are found in (2.10). Equations (2.10a) and (2.10b) are discrete kinematic equations for position and velocity, and (2.10c) comes from Newton's 2nd Law ($F = ma$) knowing neighboring nodes are connected by springs (with stiffness k) and dampers (with damping coefficients b). The term f_0 is a vector of constant forces, coming from gravity and the natural length of the springs, that do not depend on the states and thus is written on its own, and T_s is the sampling time. From here, we can easily obtain (2.11) by substitution.

$$\begin{cases} p(k+1) = p(k) + T_s v(k) & \text{(2.10a)} \\ v(k+1) = v(k) + T_s a(k) & \text{(2.10b)} \\ \quad a(k) = \dfrac{1}{m}\left(F_p(k)p(k) + F_v(b)v(k)\right) + f_0 & \text{(2.10c)} \end{cases}$$

$$\begin{cases} p(k+1) = p(k) + T_s v(k) & \text{(2.11a)} \\ v(k+1) = \dfrac{T_s}{m} F_p p(k) + \left(I + \dfrac{T_s}{m} F_v\right) v(k) + T_s f_0 & \text{(2.11b)} \end{cases}$$

ETSEIB

We use matrices F_p and F_v to apply the stiffness and damping constants to the correct nodes, considering they must receive forces only from their neighbors. Reorganizing terms, we can already see the basic structure of the space-state equations and matrices, as shown in (2.12). Specifically, we can see how matrix A, that multiples the state vector, has four distinct blocks.

$$\begin{bmatrix} p(k+1) \\ v(k+1) \end{bmatrix} = \begin{bmatrix} I & T_s I \\ \frac{T_s}{m} F_p & I + \frac{T_s}{m} F_v \end{bmatrix} \begin{bmatrix} p(k) \\ v(k) \end{bmatrix} + \begin{bmatrix} 0 \\ T_s f_0 \end{bmatrix} \tag{2.12}$$

There is however a key difference between this expression and (2.9), as here we are considering the evolution of all nodes equally, controlling none of them. The control inputs (u) force the positions of arbitrary nodes (in our case, the upper corners) to their value, and are not affected by forces coming from their neighbors. In fact, if instead of absolute positions the control inputs are increments, or displacements between time steps, they substitute the theoretical velocity depending on the neighbors. This means that, for the rows corresponding to controlled coordinates, we simply have (2.13).

$$p_i(k+1) = p_i(k) + u_j(k) \tag{2.13}$$

From a physical point of view, this checks out with $u_j(k) = T_s v_i(k)$, i.e., the displacement of the coordinate is its speed times the time it has moved. This is of course not an equation of the system, as all variables are on the same time step, and we cannot substitute the next velocity by the current displacement (over T_s). In practice, the model sets $v_i(k+1) = 0$, to help with another null entry in computations with large, sparse matrices. This is not needed by itself, given that F_p and F_v already consider the controlled nodes as special cases not affected by their neighbors, and we would end up with $v_i(k+1) = v_i(k)$, which, if we started with $v_i(0) = 0$, will mean that state is always null. In the end, the result is the same, so we can implement it in the most efficient way.

With the necessary changes on matrix A explained, the other part is then matrix B, which must relate each control input vector component j with the corresponding position vector row i. If we order all nodes in the mesh from 1 to N, and call this set of ordered indices $\mathcal{N}$, then we can call the subset of indices corresponding to controlled nodes $\mathcal{N}_C$. Then we can create a set of controlled coordinates with the expression $C_C = \{\mathcal{N}_C, \mathcal{N}_C + N, \mathcal{N}_C + 2N\}$, as the position vector is ordered to have the x coordinates for all nodes first, then the y coordinates and finally all the positions in z. This new set must have the same size as the control signals vector (in our case, for two corners, 6), and, as we needed, will relate each control signal with the corresponding row in the positions and velocities vectors: $i = [C_C]_j$. For example, if our controlled nodes are $\mathcal{N}_C = \{13, 16\}$ in a $N = 4 \times 4 = 16$ mesh (they are the upper corners if we number the nodes left to right, bottom to top), then the controlled coordinates are $C_C = \{13, 16, 29, 32, 45, 48\}$, and thus the first control signal u_1 affects row $i = [C_C]_1 = 13$, or position p_{13}. In summary, matrix $B \in \mathbb{R}^{6N \times 6}$ has only ones in positions $(i = [C_C]_j, j)$, and the rest of its elements are zero, and with these details explained and knowing C simply selects which states are outputs, we now understand what is inside the expression shown in (2.9).

Finally, it is important to note that the parameters of the model are the spring stiffness in all 3 space coordinates (k_x, k_y, k_z), the damping coefficients also in 3 directions (b_x, b_y, b_z) and an initial spring length correction factor in the vertical axis, Δl_{0z}, to counteract the fact that in linear mass-spring-damper models, we encounter the super-elastic problem, which makes the cloth model stretch vertically under its own weight. This is another contribution of this model, which avoids the common solutions involving stiffer springs (that might unstabilize the system) or nonlinearities. This makes a total of 7 parameters, that are tuned so the linear model behaves like the real cloth, but depend on mesh size and T_s.

Both models that will be used throughout this Thesis have been now presented and explained, so there is just one last bit of previous work to discuss: the initial Model Predictive Controller designed together with the linear model. Knowing the basics of MPC shown in Section 2.1, we can discuss the key points of this controller alone.

First of all, the objective function used in the optimization problem is very similar to (2.6), with an important difference: the controller introduces an artificial reference r^a between the actual desired reference trajectory and the real evolution of the states, which helps guarantee stability at a relatively low computational cost. This means the penalization done in the objective function is divided in three distinct parts: the error between the output and the artificial reference, the error between the artificial reference and the original one, and energy consumption. This is shown in (2.14).

$$\begin{aligned} J &= \sum_{k=0}^{H_p-1} \left[\left(y(k) - r^a(k)\right)^T Q \left(y(k) - r^a(k)\right) + \left(r^a(k) - r(k)\right)^T T \left(r^a(k) - r(k)\right) + u(k)^T R\, u(k) \right] \\ &= \sum_{k=0}^{H_p-1} \left[\left\|y(k) - r^a(k)\right\|_Q^2 + \left\|r^a(k) - r(k)\right\|_T^2 + \left\|u(k)\right\|_R^2 \right] \end{aligned} \tag{2.14}$$

As mentioned before, we use just index k for simplicity, but we must keep present that, rigorously, we are using the model to predict the values of variables in a time step inside the prediction horizon, while knowing the real value of them at the current time instant. This is usually written as $x(t+k|t)$, or $x_{t+k|t}$, which is the value of x at time $t+k$ while knowing its actual value at t, making the fact that we are predicting a future value from t to $t+k$, and expressing all the time indices globally instead of locally converting back to 0 for every optimization.

The weighting matrices Q, T and R give relative priority to each one of the three terms of the objective function. One common approach is to make one of them 1 to have a clear reference and make the rest either larger or smaller. If the variables being penalized differ a lot in value (errors and control inputs are orders of magnitude apart, for example), it is also common to normalize the ranges first to have a clearer proportion, with the weights really showing which part is more important. In this case, however, matrix Q was tuned with an adaptive approach, depending on the direction defined between the current position of the lower corners and the desired one at the end of the prediction horizon, with its normalized vector giving the weights for each coordinate.

The constraints for the optimization problem are the regular ones: model equations, initial state and bounds (or domains) of all variables.

The resulting MPC was implemented in simulation using Matlab [20] and CasADi [21], closing the loop with the nonlinear model shown in this section as a SOM, as seen in Fig. 2.6. The chosen sampling time was $T_s = 0.01$ s, and the dynamics showed that the COM had a settling time of around 2 seconds, which would be 200 steps, a completely unreasonable amount to be computed in real time. A brief study showed how the mean error did not significantly decrease with horizons over 30 steps, while the mean time increased progressively. The resulting simulations, done with $H_p = 30$, had very good tracking results (errors in the order of millimeters) for cloth pieces of size 30×30 cm.

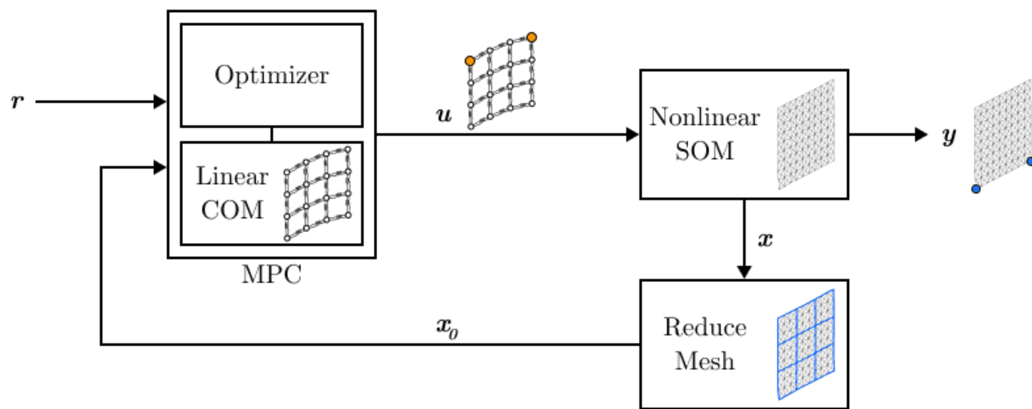

Figure 2.6: Closed-loop block diagram of MPC applied to cloth manipulation in simulation

This is the baseline closed-loop MPC implementation that was available (both article and complete codes) at the start of this Thesis, and was used as a foundation for all changes and improvements. Modifying the original code just to measure the time taken to iterate, standardize the error measurement and change the plots, we obtain the results seen in Fig. 2.7. With this trajectory and conditions, the obtained average error is about 3.15 mm.

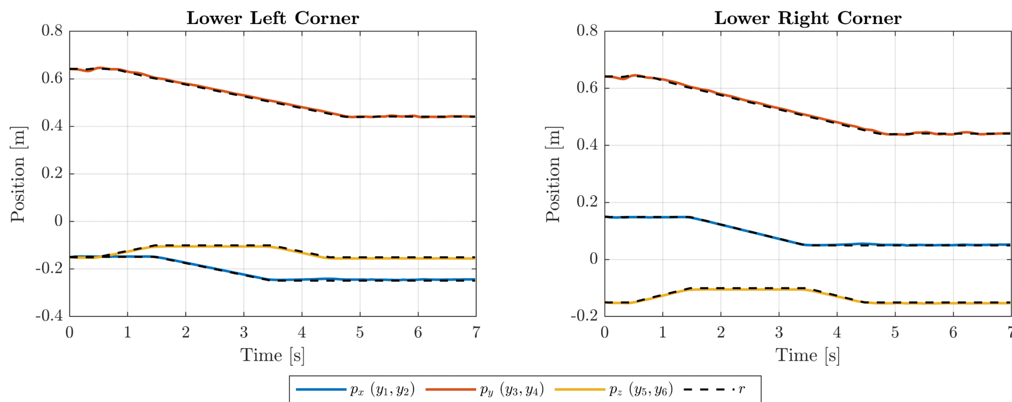

Figure 2.7: Results obtained executing the baseline MPC implementation

2.4 Controller Redesign

The provided baseline implementation proves to be valuable in many regards: it uses both the nonlinear and the linear cloth models, so the code for those is already present and available, is a first breakthrough implementing MPC with these models obtaining low errors and times, and the optimization uses CasADi in its Matlab version. This open-source tool is great to code rapid and efficient optimizations, and specifically to code MPC implementations. To top it off, it has Python and C++ versions, with quite direct equivalencies resulting in easy to translate codes between programming languages. This is a key feature needed for this Thesis, knowing the code made for simulation must be translated to implement it in a real scenario. It is worth to point out that CasADi is not a solver by itself, but can parse and connect to multiple ones, some of them built-in within the base CasADi package in all of its versions.

With that being said, the specific implementation also had severe shortcomings, the most notable one being the simulation time. Of course, in simulation, each iteration includes the optimization problem inside the controller and also the computations related to the SOM, which would not be necessary in a real case with the real system instead of a SOM. The results are impressive comparing them to the cited literature ([17] [18], with the cloth model actually included in the MPC, as mentioned in Section 2.3), reaching average iteration times in the order of tens of milliseconds, but with a time step of 10 ms, that is not enough. Even when only counting the optimization time, a trajectory of 7 s takes more than 30 s (adding all steps together) to be computed, averaging at a bit under 50 ms per iteration, which is almost 5 times the limit fixed by T_s. There are two approaches to improve this: change the optimization problem inside the controller to make it faster, or change the sampling time to have more room to compute. We will tackle both, but knowing the second one implies changes also in the parameters of the model (they all depend on T_s), this section will be dedicated to redesigning the controller as a first change.

Before continuing, another point to be addressed is the computation of the error. The provided code computed the MAE (Mean Absolute Error) for each coordinate to obtain 6 error values (2 lower corners times 3 coordinates), and then averaged them to obtain a final mean error. While this is completely valid, in this Thesis we will use the RMSE (Root Mean Squared Error), as it penalizes large errors more. The previously reported error of 3.15 mm was an averaged MAE, and the average RMSE of all coordinates goes a bit up, to 3.51 mm. Furthermore, as they are errors in the three spatial coordinates, we can compute the 2-norm to get the total position error on each time step, and then compute the RMSE along time, averaging the 2 corners at the end to have a singular value. With this method, the error of the previous simulation goes up to 6.18 mm, which is a considerable difference. Whenever an error is reported, we will always note which method has been used to obtain it, either “Average” or “Norm”.

This section will now explain the new controller design, with all the changes introduced to the optimization problem (objective function and constraints), comparing the results with the baseline case. No other changes to the code will be made for a fair comparison, resulting in the same structure and functionality. For more specific changes and improvements made progressively to bring the simulation to a point that can be implemented in a real scenario, refer to Section 2.5.

Adding an artificial reference as a step between the real state and the desired reference can be helpful, as it can differ from the input reference to guarantee feasibility and finally converge to it to enforce stability (reasoning behind its inclusion in the baseline case), but it increases complexity, with new variables for each coordinate and step within the horizon, a term in the objective function, and a terminal constraint dedicated to it. The first change to the optimization problem is to remove it completely. This removes the mentioned theoretical guarantees, but in practice the references are feasible by construction and it was also seen how the system could have unstable evolutions even with the artificial reference, because the solver has a limited time to find the optimal solution and depending on the initial conditions, some constraints would not be satisfied when giving an output. The new formulation is closer to the typical one, as seen in (2.15), where $f_{ct} = T_s f_0$.

$$
\begin{aligned}
\min_{u(k)} \ J = & \sum_{k=0}^{H_p-1} \left[\left\| y(k+1) - r(k+1) \right\|_Q^2 + \left\| u(k) \right\|_R^2 \right] \\
\text{s.t.} \quad & \left.\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) + f_{ct} \\
y(k) &= Cx(k) \\
x(0) &= x_0 \\
x(k) &\in \mathbb{X} \subseteq \mathbb{R}^n \\
u(k) &\in \mathbb{U} \subseteq \mathbb{R}^m
\end{aligned} \right\} \quad \forall k \in [0, H_p - 1]
\end{aligned}
\tag{2.15}
$$

This way of writing the new objective function J is a more compact version of (2.6), both because we have used the norm symbols instead of the equivalent expression of vector transposed times the weighting matrix times the vector again, and because we have joined both terms in a single sum, shifting the indices of the error part accordingly. This has the same reasoning as before, we have no control over $y(0)$ as the initial state is given, so there is no need to penalize its error (it will always be present), and we do not compute $u(H_p)$, as we only need until a step before to compute $x(H_p)$.

The main idea of the implementation is to have a “solver” or “controller” object defined, that wraps all the equations and relations between variables in a way that makes the computations as fast as possible, and such that it behaves like a function, to be able to call this object with the updated data during simulation (reference and initial state) and get the control signal to apply as its output. This can be done thanks to the symbolic tools from, for example, CasADi, following a defined syntax and making all variables depend on input parameters and optimization variables, the ones that we actually want to know the value of.

The next changes have to do with this specific implementation of the MPC into Matlab using CasADi, and therefore will be accompanied by code snippets showing the differences and details of how the theoretical optimization problem is actually materialized inside this “solver” object. These snippets are not full codes, and they cannot be executed independently, as they rely on variables defined on previous lines, and also can have lines omitted in between. Attaching the whole codes here would hide the important details that have been changed, so this is the best option. The resulting final version of the used codes can be found in the Appendices document found along this memory.

The first change has to do with the definition and handling of symbolic variables, and what is used as an actual optimization variable. Theoretically, both states and control signals are optimization variables, depending on each other along the horizon with the equations defined by the model. This means we would have, for N nodes, $(6N+6)H_p$ variables (the outputs are just some states). Knowing both N and H_p can be quite large, this is a big amount of variables to be handled, more if we consider that, in practice, all the states depend purely on the initial value, the matrices of the system that do not change along time, and the control signals, which are the real values we are looking for. With this reasoning, we can create a symbolic variable that represents all the state vectors along the horizon, containing explicit relations only to optimization variables and input parameters, and substitute the model constraint with it. The provided code did something similar to this, but instead defined a custom mapping function $f : (x_k, u_k) \rightarrow x_{k+1}$ with the system equations, ready to be used later, as seen in Lst. 2.1.

Listing 2.1: Original definition of the state and control variables and their relation through a function

```
% Declare model variables
phi = SX.sym('phi',3*nr*nc);
dphi = SX.sym('dphi',3*nr*nc);
x = [phi; dphi];
u =  SX.sym('u',6);

% Mapping function f(x,u)->(x_next)
f = Function('f',{x,u},{A*x + B*u + COM.dt*f_ext});

% Parameters of the optimization problem: initial state and reference
P = SX.sym('P',6*nr*nc,Hp+1);

Xk = P(:,1);
for k = 1:Hp
    % Control variables
    Uk = SX.sym(['U_' num2str(k)],6);

    % Obtain the states for the next step
    Xk_next = f(Xk,Uk);
    Xk = Xk_next;

    % [Create symbolic artificial reference ra. Add it and Uk to Opt.v Vector]
    % [Update objective function with Xk_next, Uk, ra, P]
end
```

But here we can also see how, to create this function, we need to define some temporary symbolic variables to represent x and u, that are only used internally by the function to create a link between them, and later define the actual variables used to represent the states and controls, `Xk` and `Uk`, respectively.

This definition can prove a bit confusing and overcomplicate the translation process for the real implementation. By transforming it into a matrix, we have a much simpler code, and no need for so many intermediate and local symbolic variables. The resulting code is shown in Lst. 2.2, where we also have the symbolic control signal as a matrix for convenience. There are a couple more changes done to make things easier later, such as the definition of the initial state and reference as their own variables, `x0` and `Rp`. This is not detrimental to the performance of the code, as inside we still have a reference to the original symbolic variable, `P`, which will be the input to our "solver" object. This means that every time we call the solver, `P` will have known values inside, and all the instances of this variable will get substituted automatically.

Listing 2.2: New definition of the state variables with a matrix

```
% Declare model variables
x = [SX.sym('pos',3*nr*nc,Hp+1);
     SX.sym('vel',3*nr*nc,Hp+1)];
u =  SX.sym('u',6,Hp);

% Initial parameters of the optimization problem
P  = SX.sym('P', 1+6, max(6*nc*nr, Hp+1));
x0 = P(1, :)';
Rp = P(1+(1:6), 1:Hp+1);

% Fill the state matrix so it depends only on x0 and u
x(:,1) = x0;
for k = 1:Hp
    x(:,k+1) = A*x(:,k) + B*u(:,k) + COM.dt*f_ext;

    % [Update objective function with x, u, Rp]
end
```

This modification is useful for humans reading the code and for later, but it does not change the optimization variables or their number by itself, if we compare codes. The change comes, once again, from removing the artificial reference, which was an added optimization variable to the problem too. With the states computed with the previously explained method instead of being proper optimization variables, this leaves us with just the control inputs as the only unknown variables to be optimized in the problem. This is the minimum achievable number, which in our case corresponds to $6H_p$, half of what we had, and not depending on the mesh size.

Having all the control inputs in a single matrix, it can be converted into the vector of optimization variables anywhere outside of the shown loop. With only one type of variable in this vector, the bounds are easier to apply too, as there is no need to consider which indices correspond to which variable. This is done with the few lines shown in Lst. 2.3, which in the full code are actually after line 9 of Lst. 2.2.

Listing 2.3: New definition of the vector of optimization variables and their bounds

```
% Optimization variables
w = u(:);
lbw = -ubound*ones(6*Hp,1);
ubw = +ubound*ones(6*Hp,1);
```

Right after this definition, there is also the initialization of the objective function to 0, and the definition of the constraints vector and bounds as empty variables. Removing the artificial reference also has an effect here, as the terminal constraint, which ensured the final output was exactly equal to the final value of the artificial reference, now also disappears. We can substitute it by a terminal constraint making the final state the same as the real reference at the end of the horizon, but taking a practical approach, we will not add it unless we need it, as it would increase the computational time. This means that, with the model equations implemented as explained, there are no constraints other than bounds. In fact, only bounds on the control signals remain, as together with the initial state, they will define the bounds on the whole state matrix by construction.

The final important change inside the solver is the definition of the objective function itself. As shown in Lst. 2.4, it is directly the implementation of function J from (2.15), and, of course, these three lines substitute the comment on line 16 of Lst. 2.2. It is worth pointing out that, given Matlab is a 1-based indexing language (all arrays and matrices start with index 1, as opposed to many other programming languages such as Python or C++, which are 0-based and indices start with 0), the index `k` must go from 1 to H_p in the iterative loop, meaning everything is shifted one step from the theoretical expressions, which also start at $k = 0$.

Listing 2.4: New objective function definition (for each step from 1 to H_p)

```
% Objective function
x_err = x(COM.coord_lc,k+1) - Rp(:,k+1);
objfun = objfun + x_err'*Q*x_err + u(:,k)'*R*u(:,k);
```

Besides a slight code optimization to the adaptive weight computation, and moving some hard-coded values to have them as parameters, these are all the major changes done in the controller. Lst. 2.5 shows the code to create the structures necessary to finally define the `solver` object, which actually uses the IPOPT solver [22], can be called during simulations and will return the corresponding control signals.

Listing 2.5: Creating the solver object and its settings

```
opt_prob = struct('f', objfun, 'x', w, 'g', g, 'p', P);
config = struct;
config.print_time = 0;
config.ipopt.print_level = 0;  %0-3 min-max print
config.ipopt.warm_start_init_point = 'yes';
solver = nlpsol('ctrl_solver', 'ipopt', opt_prob, config);
```

There are, however, some changes outside of the construction of the solver. First of all, the variable of input parameters, containing the reference and initial state, has been reorganized to have less zero elements, which can be important in the real implementation. Secondly, the initial guess has been changed not only to remove the part corresponding to the artificial reference, but also to give a better one to the control signals: instead of 0, the initial guess is the displacement between consecutive steps of the reference. This assumes the same movement for the upper (controlled) and lower (reference) corners, which is valid in slower trajectories, and usually a better start than 0.

We can now compare the two codes fairly under the same conditions. Three different experiments have been executed for each controller, with the resulting values for our Key Performance Indicators (KPIs), times and errors, shown in Table 2.1. Experiment 1 corresponds to the base simulation, executing the original controller as it was, and the redesigned one with the same parameters: control signals u (displacements) bounded to 0.8 mm per step, matrix Q is left as the adaptive value, and $R = 10$. Seeing that the new controller also follows the reference correctly, the bound for u was increased progressively until 5 cm per step, which is the value used for Experiment 2. Finally, R was tuned to find the best results in both cases for Experiment 3, resulting in $R = 20$ for both controllers.

Table 2.1: Comparison of KPIs between the original and the redesigned controller

Exp.	Controller	Total time [s]	Avg. time per iteration [ms]	MAE (Avg.) [mm]	RMSE (Norm) [mm]
1	Original	34.3011	49.0717	3.1479	6.1751
	Redesigned	23.5757	33.7277	2.9531	6.4159
2	Original	20.5052	29.3350	3.4077	6.8595
	Redesigned	11.6055	16.6030	3.2317	7.2620
3	Original	17.6295	25.2210	3.2210	6.6777
	Redesigned	10.9973	15.7328	2.3946	4.9087

First of all, we can see how the original controller took around 5 times the time step ($T_s = 10$ ms) in the first conditions, but it can be improved to take half that time with the tuning of Exp.3. Another important result is that the redesigned controller is consistently faster, and even if exact timing values depend on execution (background tasks running), in the optimal conditions of Exp.3 it is still around 10 ms (a whole T_s) faster than the original. This result is promising, but we must still remember 15 ms is a time step and a half, so we are still not within the range of real time feasibility. But with such an improvement, now increasing T_s and/or reducing H_p seem reasonable strategies. A new study of times and errors depending on H_p can be seen on Subsection 2.5.5, where new changes are introduced.

Of course, being faster would mean nothing if the error increased significantly, losing the remarkable tracking of the original. We can see this is not the case, as the MAEs (averaged across coordinates) are even lower in the new controller, and although the RMSEs (using the "Norm" version, explained at the beginning of this section) are a bit higher on the first two experiments, under optimal conditions it is also reduced in the redesigned controller, reaching the absolute best result under all metrics.

To finish this section, we show the results of Exp.3 in the redesigned controller in Fig. 2.8. This is the graphical demonstration that the tracking capabilities have been maintained, if not improved, with the changes introduced in this section.

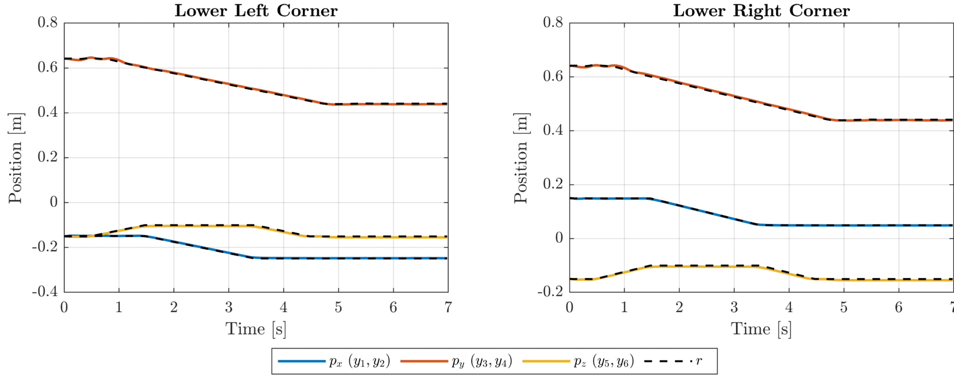

Figure 2.8: Results obtained executing the Redesigned MPC implementation

2.5 Generalizations and Improvements

After redesigning the core of the controller to make it simpler and faster, there are still some other changes needed to be done in order to get a more generalized code that can be applied to a real robot performing actual movements. This section will tackle this changes one by one in different subsections, as they do not depend on each other, even if they were implemented in the shown order, and thus sometimes benefit from changes introduced before, the order could have been different leading to the same result.

2.5.1 Accepting Models of any Size

The first change involves reducing the amount of hard-coded values in the code, specifically regarding the size of the linear model. The function responsible for creating the system matrices (A, B, f_{ct}) starts with the computation of a connectivity matrix, which is a square matrix with size $N \times N$ that indicates connections between nodes with a 1, and leaves unconnected pairs with a 0. In other words, in this matrix, each row and each column represent a node, so we can check, for example, all the connections to node in row i by finding all the columns that are 1 in this row. In principle, this matrix is symmetric, as if a node i is connected to another node j, the connection goes both ways, and both elements (i, j) and (j, i) of the matrix must be 1. While this is true for a generic connectivity matrix, in our case, we will remove some connections with the controlled nodes, to ensure their movement is governed only by the control signals (a robot moving them) and not affected by their neighboring nodes.

It is clear how this matrix depends on the total number of nodes N, as well as the proportions of the mesh, as the neighboring nodes are not the same for a 4×3 and a 3×4 mesh. For our application, we will always use square meshes, but it can be beneficial to future works based on this one to expose and code the general case, and leave the sizes as parameters. For this reason, we will talk about a mesh of n_r rows and n_c columns, with $N = n_r \cdot n_c$, instead of $N = n^2$ like in the square case.

Fig. 2.9 shows an example of a mesh of arbitrary size, chosen to be $n_r = 4, n_c = 5$ to avoid having a square for the general case. A generic node is connected to its four neighbors, but edges and corners are special cases with less connections. With the nodes numbered left to right and bottom to top, the connections of an interior node i are to the nodes on both sides ($i \pm 1$), and vertically connected nodes ($i \pm n_c$). Edge nodes, marked in a yellowish orange shade, only have three of these connections, and corner nodes (red and violet) only have two. We have colored the upper corners differently because they are the controller nodes in our application, and their connection will be only one-way, as mentioned before.

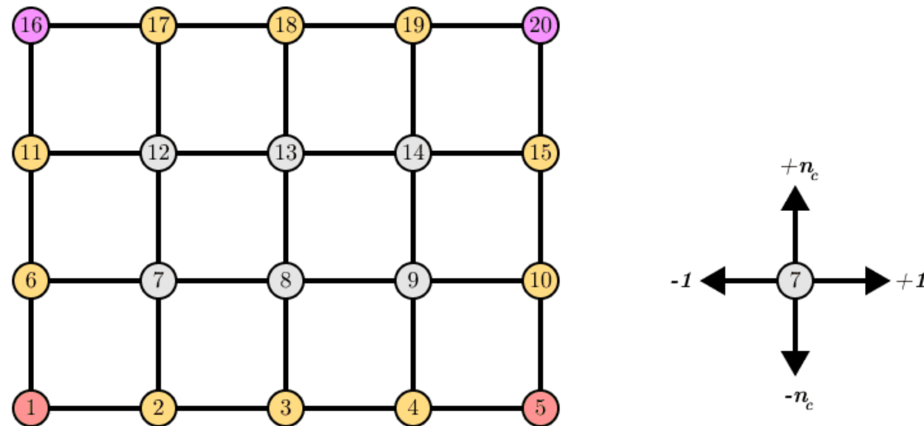

Figure 2.9: Mesh node numbering and connections (4×5 example)

With the help of this example, it is easy to deduce how to identify edge nodes, while also knowing which edge are they part of, with corners simply being part of two edges. The bottom edge is composed of nodes the indices of which are lower or equal to n_c. Analogously, the top edge has all the nodes with indices greater than $(n_r - 1)n_c$. The right edge has all the nodes with indices divisible by n_c, and if we divide the indices of the nodes in the left edge by n_c, the remainder is always 1. This can be deduced from a more formal identification of the right edge, as those all have remainder 0.

This means that, instead of hard-coding a matrix of zeroes and ones that is only valid for a fixed mesh size (in the original case, 4×4), we can build a connectivity matrix for any mesh with a few lines of code, as shown in Lst. 2.6.

Listing 2.6: Creation of the connectivity matrix for a mesh of any size

```
1  conn = zeros(nr*nc);
2  for i=1:nr*nc
3      if(mod(i,nc) ~= 1), conn(i, i-1) = 1; end
4      if(mod(i,nc) ~= 0), conn(i, i+1) = 1; end
5      if(i > nc),         conn(i, i-nc) = 1; end
6      if(i <= (nr-1)*nc), conn(i, i+nc) = 1; end
7  end
8  conn(nctrl,:) = 0; % Remove connections on controlled nodes
```

As shown in the last line, once the actual connectivity is built, we set all the rows corresponding to controlled nodes (set $\mathcal{N}_C$) to 0, indicating their neighbors have no effect in them. However, it is very important to remark that the controlled nodes do affect their neighbors when moving, so while, for example, element $(16, 11)$ is zero in the mesh of Fig. 2.9, element $(11, 16)$ will be 1.

On the construction of the model, this connectivity matrix (M_C) is then used to apply the spring and damper forces to neighbor nodes. In steady state, the sum of forces on all nodes must be zero, so we can create a new "unitary force" matrix (F), where the diagonal element of each row is the result of adding all the columns from the connectivity matrix together, and then we subtract the connectivity matrix itself, so the sum of all the elements of any row is 0, as shown in (2.16). This matrix is then used to define the previously mentioned F_p and F_v, multiplying all the elements by the stiffness or damping in the corresponding coordinate. In the construction shown in (2.17), each block is a square matrix of size N, and the code had to be changed from the specific $N = 16$ to a general case.

$$F \longrightarrow \begin{cases} F_{i,i} = \sum_{k=1}^{N} M_C(i,k) \\ F_{i,j\neq i} = -M_C(i,j) \end{cases} \tag{2.16}$$

$$F_p = \begin{bmatrix} k_x F & 0 & 0 \\ 0 & k_y F & 0 \\ 0 & 0 & k_z F \end{bmatrix}, \quad F_v = \begin{bmatrix} b_x F & 0 & 0 \\ 0 & b_y F & 0 \\ 0 & 0 & b_z F \end{bmatrix} \tag{2.17}$$

The same reasoning applies to the creation of system matrix A, which apart from using F_p and F_v, as shown in (2.12), it also has identity matrices of size $3N$, and this dependency must be explicit, not a hard-coded number. Additionally, for both A and B, the set of controlled coordinates is now introduced as a parameter of the linear model, and is not just defined as the upper corners with the indices depending on size, but is left open in case this code is used with an application where other nodes are the ones being controlled. Lst. 2.7 shows this new code, which also materializes (2.13) and the considerations explained in the paragraphs immediately after.

Listing 2.7: New code to create matrices A and B

```
A = [eye(3*N)          Ts*eye(3*N);
     (Ts/m)*Fp    eye(3*N)+(Ts/m)*Fv];
A(cctrl, 3*N+1:end) = 0;
A(cctrl+3*N, :) = 0;

B = zeros(2*3*N,6);
for i=1:length(cctrl)
    B(cctrl(i), i) = 1;
end
```

Finally, the computation of f_{ct} is also done depending on mesh sizes: gravity affects the rows corresponding to v_z for any size, the force corresponding to the initial length is set to the correct coordinates, and the rows of controlled coordinates (matching an index of C_C) are set to zero. This completes the changes to create a linear model of any size. The only other change from hard-coded indices to parametric ones is in the function to take a reduced mesh, which now can take the original and the final sizes as parameters, and selects the correct nodes to keep.

2.5.2 Using a Local Cloth Base

Another issue of the simulation comes from the definition of the parameters of the linear model and the possible movements to be executed in consequence. As mentioned in Section 2.3, the linear model has seven parameters in total, with the first six being stiffness and damping in all three space coordinates. As one would expect by the properties of a thin textile material such as the one being considered, the stiffness in the direction perpendicular to the cloth plane (when it is extended) is much lower than in the two axes of that plane, and the same can be said about the damping, as seen in Table 2.2, where the cloth is in the xz plane, and the y component is much lower in both stiffness and damping.

Table 2.2: Original linear model parameters, tuned using the nonlinear one

Var.	x	y	z
k	-305.6028	-13.4221	-225.8987
b	-4.0042	-2.5735	-3.9090
Δl_0	-	-	0.0312

This difference has serious implications, as the cloth orientation has a direct effect on the parameters. As fast proof, we can rotate the original trajectory 90 degrees, along with the initial mesh position, while keeping the parameters in the same order (or vice versa). The results can be seen in Fig. 2.10, and it is clear how, with this implementation, rotations are not feasible, as they result in an unpredictable evolution.

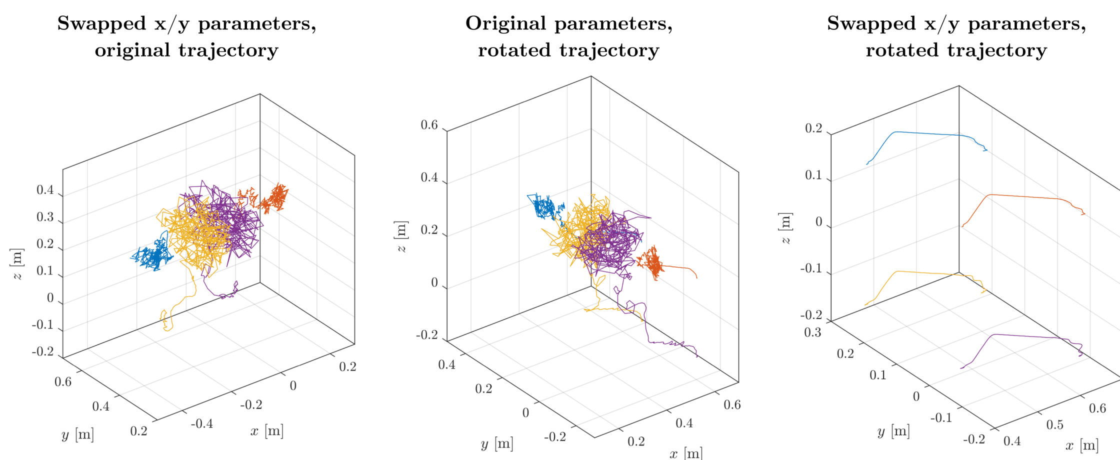

Figure 2.10: Corner trajectories rotating the cloth trajectory and/or parameters

ETSEIB

Nevertheless, there is a promising result. We can see how rotating the trajectory and also swapping the order of the parameters results in correct tracking, as shown in both the last plot of Fig. 2.10 and in Fig. 2.11. The tracking error and computation time are both still on the same range as before, so this leads us to the solution: rotating the parameters so they always match with the orientation of the cloth.

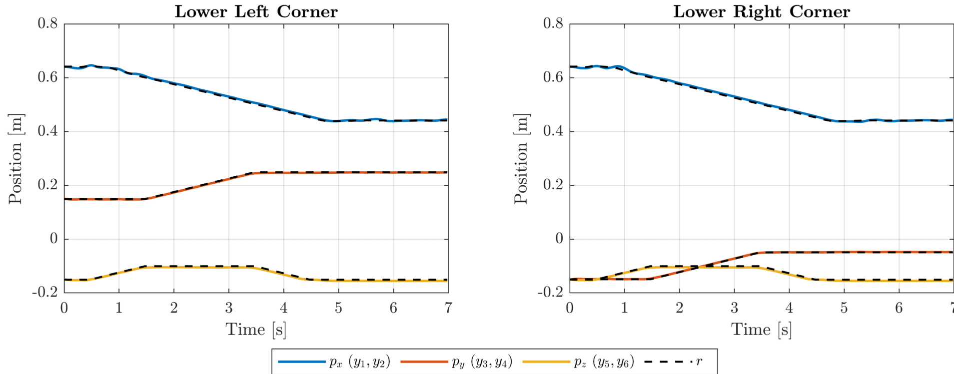

Figure 2.11: Evolutions obtained rotating the trajectory and swapping the parameters accordingly

A first option we could consider is, knowing the orientation of the cloth, having the parameters in local coordinates, and projecting them into the global axes. Following the diagram of Fig. 2.12, where the cloth is represented as a black line, as seen from a top 2D view, for simplicity. In (a), we have a large parameter in x (red) and a small one in y (green), matching the global axes. If the cloth rotates like in (b), these parameters are now also rotated the same amount (now magenta and orange, because they are not in the global x and y directions), but we can project them into the global base components. In (c), these projections are recolored to match the original red and green, to show the projections of matching colors will be added together. Finally, in (d), the resulting projections are shown, which in fact correspond to the projections of the sum of both vectors. For example, for a rotation of angle θ, we would have $k'_x = k_x \cos\theta + k_y \sin\theta$.

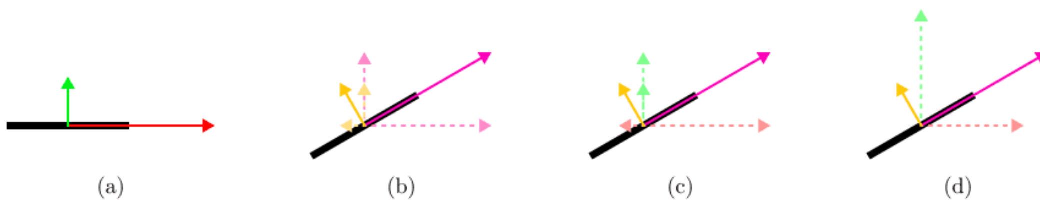

Figure 2.12: Top view of a cloth piece and representation of local parameters and projections

With this process, however, we lose information. Even if in rotations multiple of 90 degrees the parameters would align with their projections and be exactly right and rotated, which would be an improvement from the base case, in the shown example, both resulting parameters in global axes would have relatively large values, which goes against the point of always knowing which is the direction with the minimum stiffness (in absolute value, perpendicular to the cloth) to simulate a correct evolution of the COM, especially if it moves in this direction.

The proposed solution is to implement a local cloth base and, instead of converting the parameters to global axes, do the opposite, and transform all the necessary data into the local base, to simulate the evolution knowing the cloth is always in, for example, the local xz plane.

Now the question is how to define this base so that it is intuitive, useful, and can be computed online with the available data to convert to and from it. To tackle this, we can think about what can we access during executions. In simulation, we have access to the entire SOM state vector (positions of all the mesh and velocities of all nodes), the control signals (which are increments, but added to the starting point, they are also the positions of the upper corners), and the reference trajectory. Even in a real implementation, we will have access to the reference, the control signals, which we can convert to global positions cumulatively adding them to the initial state of the controlled nodes, and, with a sensor like a camera and some processing code, it is reasonable to think that we can also have the positions of all the nodes in the real cloth.

To work with the minimum necessary first in case something does not work as well as expected, we can try to build a cloth base using only the control signals converted to absolute positions. As shown in Fig. 2.13, this is actually possible. The first step (a) is to define the local X vector in the direction from one corner to the other, $X^{cloth} = [u_2, u_4, u_6]^T - [u_1, u_3, u_5]^T$. The other two axes must be contained in the plane perpendicular to this axis, as shown in (b). Without using feedback data (full mesh), we could use the reference to get a guess of where the lower corners are, or at least should be, and define the local vertical axis with the direction between lower and upper corners. To simplify even more, we can directly assume gravity will pull the lower corners down, and force the local Y^{cloth} axis to have 0 vertical (absolute Z) component, as shown in (c), which also defines the local Z^{cloth} axis as the cross product between the other two, $Z^{cloth} = X^{cloth} \times Y^{cloth}$, as seen in (d).

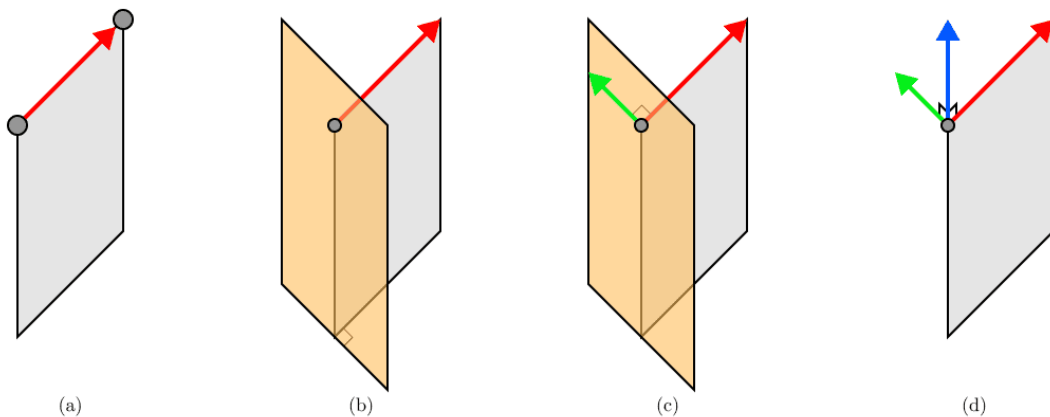

Figure 2.13: Process to obtain the local cloth base

This approach has the advantage of being the easiest to compute, requiring the minimum amount of data, and the most reliable one, the control signals that we are computing and introducing to the system, not the reference that might not be exactly tracked or feedback which might have noise producing great inaccuracies in this precise computation. It also guarantees orthogonality between axes by construction,

which is not achieved by using the other mentioned methods. For example, if the upper corners are at different heights, but the cloth is not moving and the lower corners are roughly right under the top ones anyway, as seen in Case 1 of Fig. 2.14, if we define the Z^{cloth} axis using information from the lower corners (either reference or feedback), it will be vertical and up, which is not perpendicular to the X^{cloth} axis (orange plane, which in this 2D view is just a line). This means that we would have to check perpendicularity between vectors and project into the perpendicular plane if needed, which means more operations and special cases to be considered if we compare to the simple method, which would define the Y^{cloth} axis first, horizontal and perpendicular to X^{cloth}, and then the last axis would be perpendicular by construction.

Figure 2.14: Worst cases for the two considered methods of computing the cloth base

This does not mean the minimal method comes without issues, as we can see in a situation like Case 2 in Fig. 2.14, where both upper corners at at the same height, but the cloth is accelerating or decelerating, and forms a curved surface in space (a curve in this 2D side projection). Here, using only the upper corners, the Z^{cloth} axis would simply be pointing upwards, with Y^{cloth} pointing horizontally left. This is a very rough approximation of a cloth "plane", which is much better achieved considering the lower corners, and defining the Z^{cloth} axis as the dashed blue line in the figure. We must remember the main objective of this base change is to have the Y^{cloth} axis as close as possible to the perpendicular direction of the cloth, to match the direction with the parameter that differs the most from the rest. However, in a real implementation, the lower corner information might not be accurate, as the reference possibly does not consider this "lag" (if accelerating) or "rise" (if decelerating) in the lower corners, and the feedback might be too slow or imprecise to have an accurate approximation. This simple method also has issues when X^{cloth} is completely vertical, and does not consider rotations along this axis either, but in the studied reference tracking situations, these situations will not be found.

In the end, for the real implementation, the simplest method will be used at first, with the sequence shown in Algorithm 1. However, if new trajectories and scenarios prove it too inaccurate, we can swap to a method using lower corner information without major changes. To prove the new movements possible, we can simulate a different trajectory involving rotations, as shown in Fig. 2.15. Needless to say, executing this trajectory without the base change results in a chaotic evolution similar to the first ones of Fig. 2.10.

Algorithm 1 New sequence to call the MPC solver in local coordinates

Require: $r,\ u^{prev} = x_0[C_C],\ R^{cloth} = R_0^{cloth}$

1: **for** $tk = 1$ **to** $tk = \text{length}(r)$ **do**
2: $\quad x \leftarrow \text{GetFeedback}()$ ▷ Update from SOM or real cloth
3: $\quad r_{Hp} \leftarrow r(tk : tk + H_p)$ ▷ Sliding window
4: $\quad r_{Hp}^{cloth} \leftarrow (R^{cloth})^{-1} \cdot r_{Hp}$ ▷ Rotate all reference points
5: $\quad x^{cloth} \leftarrow (R^{cloth})^{-1} \cdot x$ ▷ Rotate reorganizing variables as needed
6: $\quad u^{cloth} \leftarrow \text{SolverMPC}(x^{cloth}, r_{Hp}^{cloth})$ ▷ MPC uses COM parameters: local base
7: $\quad u \leftarrow R^{cloth} \cdot u^{cloth}$
8: $\quad u^{abs} \leftarrow u + u^{prev}$ ▷ Absolute corner positions
9: $\quad u^{prev} \leftarrow u^{abs}$
10: $\quad X^{cloth} \leftarrow \begin{bmatrix} u_2^{abs} - u_1^{abs} & u_4^{abs} - u_3^{abs} & u_6^{abs} - u_5^{abs} \end{bmatrix}^T$ ▷ (“Right” - “Left”) upper corners
11: $\quad$ **if** $[X_1^{cloth}, X_2^{cloth}] \neq 0$ **then** ▷ If X^{cloth} is vertical, keep Y^{cloth}
12: $\qquad Y^{cloth} \leftarrow \begin{bmatrix} -X_2^{cloth} & X_1^{cloth} & 0 \end{bmatrix}^T$ ▷ Guaranteed orthogonality
13: $\quad$ **end if**
14: $\quad Z^{cloth} \leftarrow X^{cloth} \times Y^{cloth}$
15: $\quad R^{cloth} \leftarrow \begin{bmatrix} X^{cloth}\|X^{cloth}\|^{-1} & Y^{cloth}\|Y^{cloth}\|^{-1} & Z^{cloth}\|Z^{cloth}\|^{-1} \end{bmatrix}$ ▷ Normalize
16: $\quad \text{ApplyControl}(u)$ ▷ To SOM or real cloth
17: **end for**

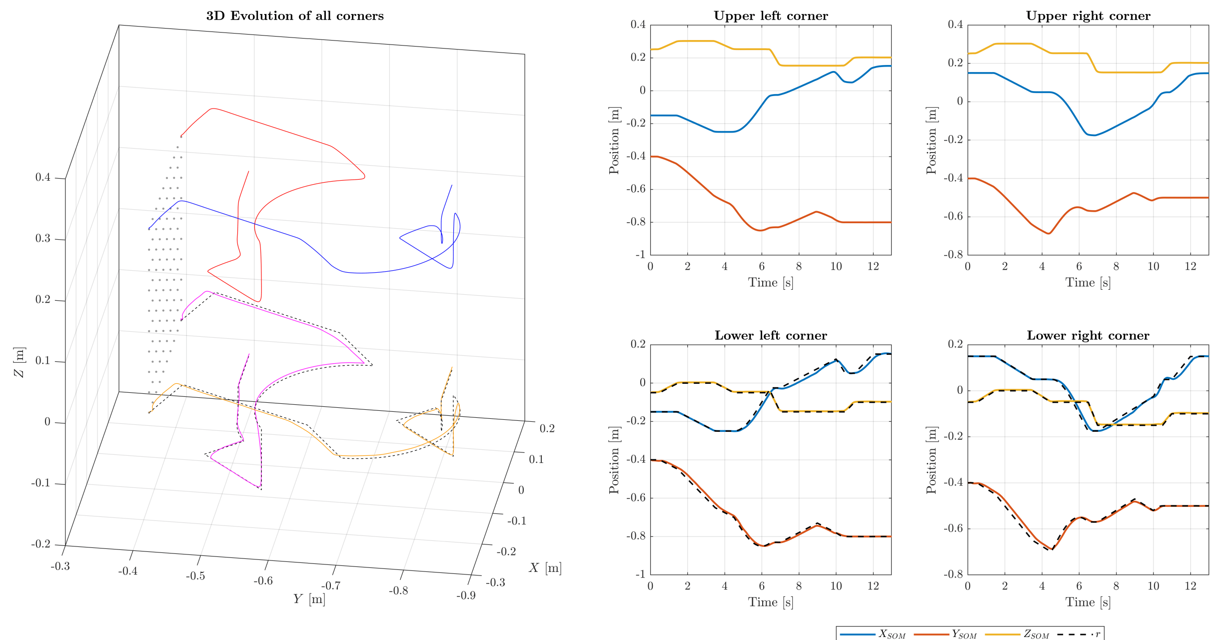

Figure 2.15: Results for a new trajectory with rotations, using a local cloth base

2.5.3 Outputting a Single TCP Pose

The considered setup includes two independent robotic manipulators, so each one picks and controls an upper corner of the cloth. While this is an ideal case and allows to execute varied tasks while tracking a reference for the lower corners, as a first real implementation, it is better to reduce the complexity of the system and start with a single robot controlling the cloth piece.

This has two important implications: the two controlled corners are no longer independent, and we cannot send the computed control inputs directly to the real robot, as its TCP will not be in a cloth corner. The diagram seen in Fig. 2.16 shows a possible solution, adding a rigid piece that connects both upper corners and has a new point where the robot can be connected (orange circle).

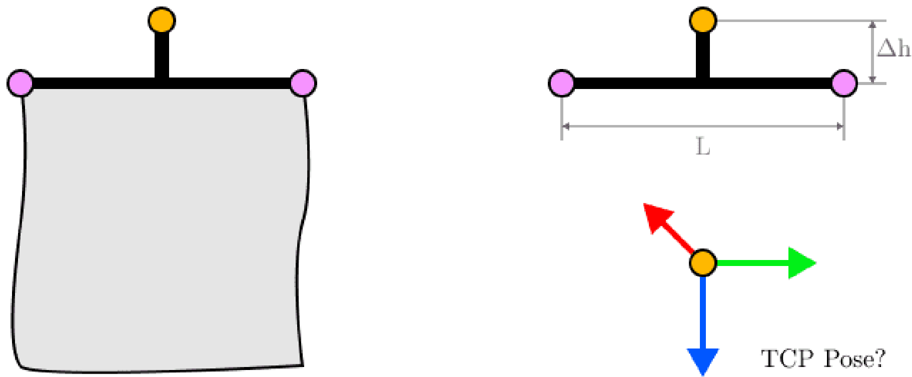

Figure 2.16: New setup with a rigid piece between the TCP and upper corners of the cloth

This piece forces the constant distance between upper corners (L, the width of the cloth, or just its side in a square case), which will also limit the possible trajectories for the lower corners that we can track, and adds a small offset (Δh) to the new connecting point. In the shown situation, this distance is vertical, but if the cloth changes orientation, this offset must be added in local cloth coordinates. This brings us to another consideration, to move a robot arm to a desired point in space, we need a full pose for the TCP, including position and orientation. Before being able to rotate, we could have fixed this orientation to an arbitrary value so the gripper always pointed down, but now we must also compute this orientation, which will clearly be related to the local cloth base.

First of all, however, there is a core change that must be analyzed, as a result of the constant distance between upper corners. It is clear that the set of feasible control signals is now reduced, or in other words, constrained to those that keep this distance constant. Given the control signals themselves are increments, it is easier to force this relation to the absolute positions of the upper corner nodes in the state vector, which, with our implementation, already contains the equations necessary to relate their values to the initial parameters and the control signals (optimization variables). Computing a distance or a 2-norm involves quadratic operations, even if we square it to remove the square root as seen in (2.18).

$$\|d\| = \sqrt{d_x^2 + d_y^2 + d_z^2} = L \implies \|d\|^2 = d_x^2 + d_y^2 + d_z^2 = L^2 \tag{2.18}$$

This is very important, because adding a constraint of this nature to our previously defined optimization problem (2.15) converts it from a Quadratic Program (QP), where the objective function is quadratic but the constraints are linear, to a Quadratically Constrained Quadratic Problem (QCQP) [23], which is more complex. Luckily, this is one of the simplest nonlinearities, and still a special case where convexity still applies, and thus computations can still be quite fast.

The specific implementation with CasADi and Matlab is shown in Lst. 2.8, which is introduced on the definition of the "Solver" object, right after line 14 of Lst. 2.2. Variable `g` is the vector of constraints, which need to be satisfied within some bounds. As we have subtracted the constant distance squared in the constraint vector, this should be an equality constraint, with `gbound=0`, but we can keep it as a variable in case there is a need to relax the constraint to reduce computational time.

Listing 2.8: Additional quadratic constraint to ensure constant distance

```
1 x_ctrl = x(cctrl,k+1);
2 g = [g; sum((x_ctrl([2,4,6]) - x_ctrl([1,3,5])).^2) - lCloth^2 ];
3 lbg = [lbg; -gbound];
4 ubg = [ubg;  gbound];
```

The output of the solver is still a 6-component vector with displacements corresponding to all three coordinates of both upper corners. Adding the displacements together from the start and with the initial state of these points, we can obtain the absolute positions of the upper corners on every step (which we previously called u^{abs}). With them, the center point is easily computed with a mean, and then we only have to add the offset Δh to get the TCP position. As mentioned before, this cannot be done directly in global coordinates, as the cloth (and rigid piece) might be in any orientation. We would not even consider this without the change introduced in Subsection 2.5.2, which also provides us an easy way to convert to a local cloth base with R^{cloth}. The complete process to obtain the TCP position is shown in (2.19).

$$\Delta p^{cloth} = \begin{bmatrix} 0 \\ 0 \\ \Delta h \end{bmatrix} \rightarrow \Delta p = R^{cloth} \cdot \Delta p^{cloth} \rightarrow p_{TCP} = \frac{1}{2} \begin{bmatrix} u_1^{abs} + u_2^{abs} \\ u_3^{abs} + u_4^{abs} \\ u_5^{abs} + u_6^{abs} \end{bmatrix} + \Delta p \qquad (2.19)$$

The only part remaining to output a full pose is the TCP orientation. It can be expressed in multiple ways, for example a rotation matrix like R^{cloth}, a quaternion or with Euler angles. Quaternions are the most extended implementation given their compactness while keeping uniqueness and non-singular properties [24]. Methods to convert to and from rotation matrices are available in libraries for common coding languages, so they will be used having the final real implementation in mind. The process to obtain the TCP orientation is as follows: knowing the TCP base from the robot side has its Z axis pointing outwards from the arm, we will adapt the local cloth base to match this, which in fact means inverting the Z^{cloth} axis, pointing from the cloth towards the robot. We could then arbitrarily choose to keep one of the other axes the same as in the cloth base, and invert the final one to complete a base with right-handed (standard) orientation. However, after checking with the real setup, it is more convenient to build the

TCP base as $R^{tcp} = [Y^{cloth} \quad X^{cloth} \quad -Z^{cloth}]$ (this matches the axes shown in Fig. 2.16). Finally, we convert this rotation matrix into a quaternion (with an off-the-shelf function) to obtain the full TCP Pose.

In summary, with just the control signals on each step, we can compute the absolute positions of the upper corners, and with those we can get both the position and orientation of the TCP.

2.5.4 Using other Models

During the course of this Thesis, and parallel to it, there were some updates in the implementation of the nonlinear model, meaning there was a need to update its use in the simulations, and also compare the new version with the linear mass-spring-damper model, to check the linearization still holds and they have similar dynamics. Furthermore, while starting the real implementation and checking the rates of the different parts of the control scheme, the vision feedback proved to be the slowest part, creating a need for a "backup" fast model to fill in the gaps between real feedback data. To be the fastest possible, the model must be linear, so the simulation code must be adapted to also work with a linear SOM. This subsection is dedicated to these two changes.

The new nonlinear model is quite similar to the old one, with a few parameter adjustments. The main difference is found in the functions used to simulate a step. The old codes had first and second order simulators, needing one or two previous states, respectively, and the simulations with this model as SOM used the second order step simulator. With the new update, the first order simulator is improved, and the second order one is removed. Apart from that, there was no independent function to initialize the model on its own, and in demanding or chaotic scenarios, the simulation got stuck trying to minimize an error than never went down. To enable the use of this new model, each one of this issues had to be addressed.

As a first change, a timeout was introduced on the loop that simulates a step. Analyzing how long a regular simulation took, and then trying different values, a final threshold was set to 0.1 s. This updated simulation function used new variables and a different nomenclature, so a "parser" function to translate between names was created. Then, to have an independent function that initializes the model as its own object and nothing more, parts of the code were separated and adapted to reach that point. The calls to these functions can be seen in the codes shown in the Appendices document, and even if the functions themselves are not included, a link to the entire source code is provided there.

Changing the nonlinear model, even if slightly, means the comparison with its linear counterpart must be done again, to check it still stands, and to what degree. We can initialize both models on the same position and enter the same control inputs, to check how they evolve. Starting with a movement of 10 cm in only one axis and completed over 1 s (a speed of 10 cm/s with $T_s = 0.01$ s means a displacement of $u = 1$ mm each step), the two models behave almost identically for all axes and directions. On constant velocities, including with no movement, the linearization seems to perform the closest to the nonlinear model, and the notable differences happen when accelerating, decelerating or changing direction. This means that there is not a limit for the prediction time ($T_s \cdot H_p$) fixed by the similarity between models, but there is a limit on how fast these movements can be, or more accurately, change.

To demonstrate this, a combined step input was created, with movement in all three axes. The upper corners must move 10 cm in x, 30 cm in (negative) y, and 20 cm in z. In Fig. 2.17, this distance is set to be travelled within 1 s. We can see how the inputs change instantly to the corresponding displacement per step values, and back to 0 when a second has passed. The evolutions of the lower corners are almost identical between models (solid lines for the nonlinear, NL, and dashed for the linear, L). Changing the movement time to 0.2 s results in the evolutions shown in Fig. 2.18, where the inputs are five times larger (maximum of 15 mm per step), and the lower corners of the linear model present oscillations more pronounced than those of the nonlinear one. Even if they stabilize to the same steady state after a few seconds, the differences between both models can be significant. The results are worsened by the start and stop being so close together, as keeping the same speed for longer and then stopping, or even fast speeds with progressive accelerations, give better results, as shown in Fig. 2.19.

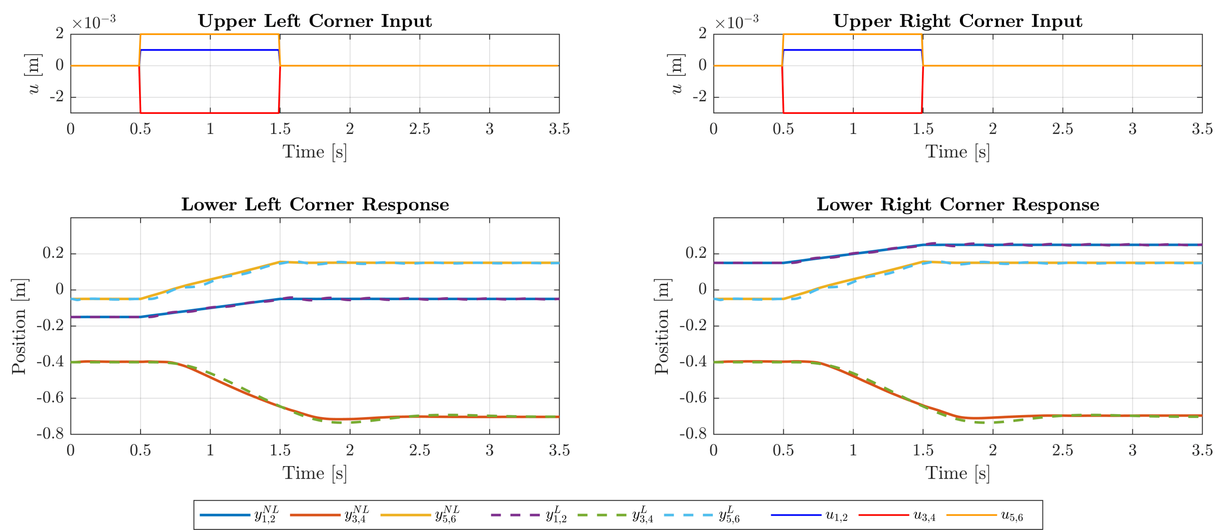

Figure 2.17: Comparison between the new nonlinear model and the linear one (1 s movement)

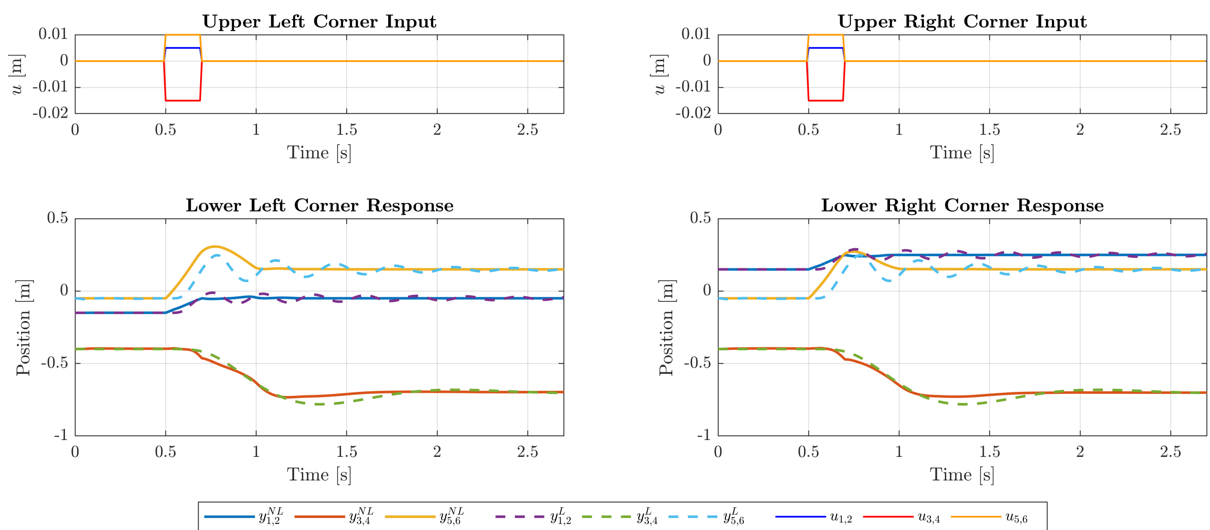

Figure 2.18: Comparison between the new nonlinear model and the linear one (0.2 s movement)

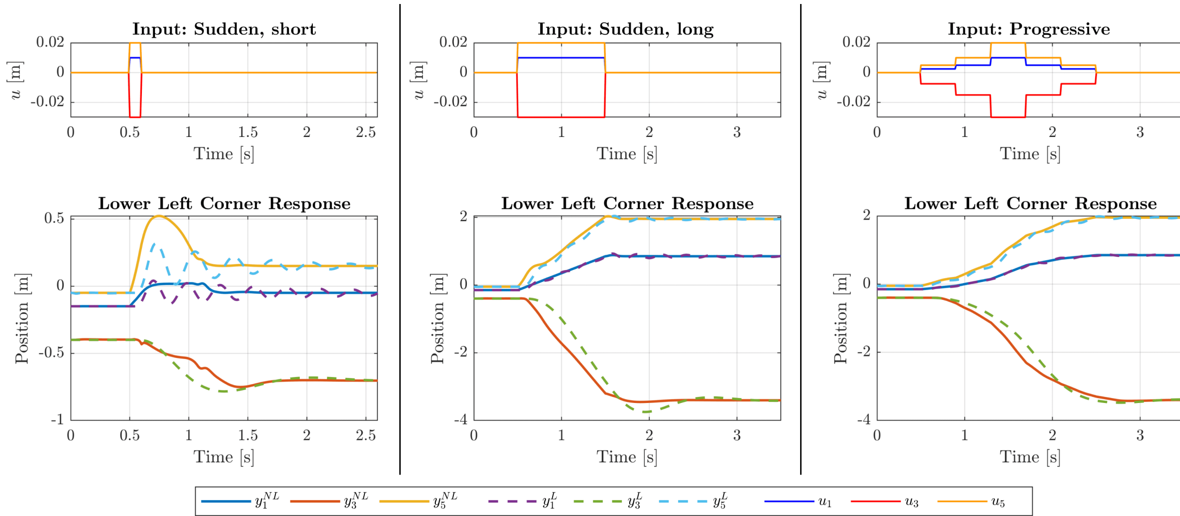

Figure 2.19: Comparison between three scenarios with the same maximum speed

These results prove how the limitations to consider are not the maximum prediction time nor the maximum displacement (pure bound of u), but how the displacement values change from one step to the next. The change introduced in Subsection 2.5.5 ties with this idea.

While changing the function that initializes the nonlinear model, some extra changes were introduced. Before, the mesh was always initialized in a fixed position of space, with the cloth plane being xz. It made sense, given how rotations were not allowed, but now, if we want to try new trajectories starting on other points, this limitation must be removed. The new process to initialize the model is to first load the desired reference trajectory, and then deduce the implied parameters from it, which are: cloth side length, initial cloth center and angle.

The cloth side length is defined by the distance between the first points of the reference trajectories for each of the lower corners (even if their distance is not fixed like on the upper corners, the range of possible movements is now limited, and at the initial position, we will always have the cloth extended), its initial center is halfway between the lower corners and half a side length up, because all trajectories will start with a vertical cloth, and thus the only allowed rotation will be along the global Z axis, with an angle that can be obtained with an arctangent.

These three values are then given to the nonlinear model initializer function, which creates a mesh with size and position according to them. A function to create a mesh for the linear model was created too, and if the initial angle is different from zero, the local base is updated accordingly, defining the COM in local axes so the stiffness and damping parameters are applied in the correct directions.

Setting this new nonlinear model aside, looking at a possible real implementation, there was a need for a way to keep the execution going even without constant feedback, given how it is the slowest part of the control scheme, returning data after several samples. The easiest solution is to have a backup model, a SOM even in a real scenario and not in simulation, to update the initial states between feedback updates.

This backup model needs to be as fast as possible, because it will have to be updated every step, adding computational time to a controller that already takes longer than a time step. Thus, the model must be a linear one, of the minimum size possible (but not smaller than the COM), and it will be updated on every step in the real case as if it was a simulation. To prepare this case, we can alter our closed-loop simulation to work with a linear SOM, as an intermediate step that will simplify the real implementation.

The definition of the controller is exactly the same, and the first half of every iteration during simulation is also unchanged: get a sliding window of the reference, rotate it and the initial state of the COM, call the solver, and obtain an increment in local coordinates, which we can convert back to the global base and add with previous data to obtain the absolute positions of the corners and the TCP pose, as seen in Subsections 2.5.2 and 2.5.3. However, now the SOM is also linear, so instead of using u or u^{abs} to update it, we must simulate a step using local coordinates too. The process involves taking the previous state in global coordinates, rotating it (positions and velocities), then applying the linear system equation (with new dedicated matrices $A_{SOM}, B_{SOM}, f_{SOM}$) to obtain the next states in local coordinates, which we then rotate back to the global base to save the evolution and update the initial COM state.

It is clear how the intention is to still have slightly different models for COM and SOM, even if they are both linear. For example, if the feedback gives a mesh of side size 13 ($N = 13^2$), we can have a SOM of side size seven and a COM of side size four, as they can both be obtained by taking a reduced version of the largest one, as seen in Fig. 2.20. In fact, for square meshes, we can use $n_{big} = k(n_{small} - 1) + 1$ to find the bigger sizes that can be reduced into a given small size. However, changing the size of a linear model will modify its parameters, as, for example, spring forces depend on length and the nodes will be at different distances, and there is no given expression to relate size and parameters. Even worse, the method used to match the behavior of the linear model with the nonlinear one is based on an optimization, which reaches the maximum number of variables or times out just with models of side size 7, even using shorter trajectories to compare evolutions.

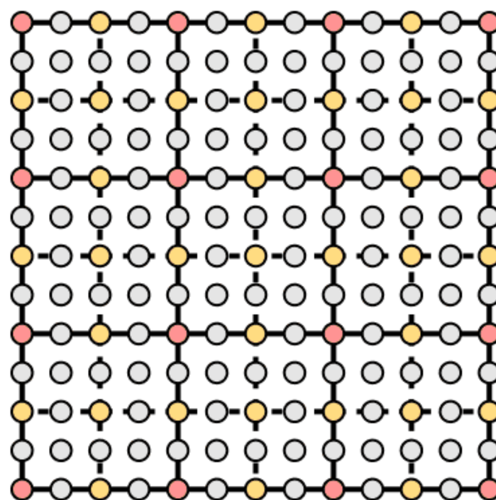

Figure 2.20: Large 13×13 mesh with nodes taken on reduced 7×7 and 4×4 meshes highlighted

And that is not all, parameters of the linear model also depend on the sampling time (T_s). This means we cannot change this value at all without having to find the corresponding parameters, which can only be done for a model of side size 4 and even then the optimization can fail to converge and the behavior of the linear models ends up not being similar to the nonlinear one. To prove how much this limits us, two scenarios where the model parameters change have been simulated in open-loop, keeping the original values, and are shown in Fig. 2.21. The first one corresponds to changing the COM side size from 4 to 7, and input a very slight movement in z. Even before starting the movement, the cloth falls under its own weight, causing a reaction that unstabilizes the system. The second case, on the right, is changing T_s from 10 to 12 ms, and even just that slight change produces an oscillation in z that instead of disappearing, suddenly increases again.

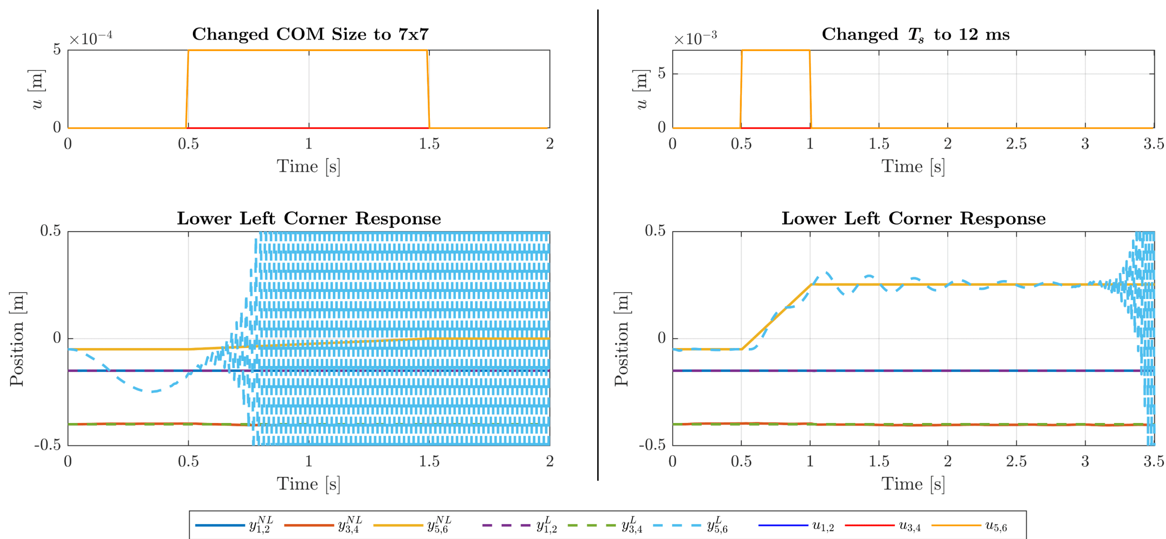

Figure 2.21: Unstable evolutions when changing model size or sampling time but not the parameters

These results show how there is no way of using the original model parameters in a general case, and together with the limitations of the original method to tune them, justify the search for a new technique to obtain them for multiple sizes and time steps, for example, one involving learning algorithms. This is the motivation behind what will be explained in Section 3.2. Once the parameters for different conditions have been found, we will be able to apply them to our new codes.

As a last note regarding the implementation using a linear SOM, if the model is exactly the same as the COM, and everything is done in simulation so there is no disturbances or noise, it is logical that both models will evolve in exactly the same way. In simulation, this can mean getting to the point of computing the whole control sequence from the start, and then applying the corresponding pre-computed control signal on every step. This is of course going back and against the basis of MPC and its receding horizon, which is implemented as explained because real systems do not behave perfectly and exactly as their models and predictions, so it does not make much sense. However, in a real implementation, we could explore this special case, and reduce the computational load not calling the optimizer unless we receive a feedback signal. Once we do and predict enough steps, the “backup” SOM will evolve exactly as predicted, so we can keep applying the computed control inputs until a new feedback signal arrives.

This approach will be left as a last resort in case the optimization takes longer than a time step in the real implementation and cannot be called on every iteration. Theoretically, it should have the same behavior as increasing the sample time to values multiple of the considered one, which cannot be done in our system without changing the cloth model parameters, but it is not commonly found in literature. For the simulation code, the general case accepting a linear SOM different than the COM will be kept, even if before applying learning algorithms we can only use the exact same model for both, which of course produces almost perfect results, with close to no tracking error, and low times even adding the time taken to simulate the SOM evolution (still over 10 ms per iteration, but under 20 ms).

2.5.5 Minimizing the Slew Rate

Formulating the optimization problem to be solved in the controller, the two terms of the objective function (minimizing errors, $y - r$ and control signals, u) were justified to improve tracking while not having disproportionate energy consumption, as mentioned in Section 2.4. However, in a real implementation, a robot will consume energy also while staying in place, as the motors must produce the necessary torque for gravity compensation, i.e., making the robot not fall under its own weight, so the consumption is not directly related to the absolute value of the control signals, which are displacements. This section explores a suitable alternative for the term related to u in the objective function, and analyzes the implications of this change.

The control inputs u of the considered scheme are the upper corner displacements in a time step, as seen in (2.13). They are closely related to the speed of the upper corners, as with a fixed T_s, they tell the distance to be traveled in this amount of time. While reducing speed can have its benefits, especially having an upper bound to avoid really fast movements in an environment that will probably have humans nearby, this does not justify their inclusion in the objective function to minimize energy consumption, as a fast movement at constant speed and in a straight line can consume less than one constantly changing speed and direction. This hints at acceleration being a better variable to be minimized, or more precisely the rate of change of the displacements represented by the control signals, rather than the displacements themselves or the absolute positions of the corners. For a generic control input u, the rate of change can be written as Δu, following (2.20), and this new variable is commonly known as slew rate [25].

$$\Delta u(k) = u(k) - u(k-1) \tag{2.20}$$

Penalizing slew rates instead of control signals has clear advantages in the studied problem. Faster speeds are not considered worse by themselves anymore, but sudden changes in speed are. Starting or stopping a movement is now penalized, but the entire cruise time has no effect if done at constant speed, and, given the quadratic nature of the cost function, it is worse to have an instant change of velocity than to progressively alter it (e.g., $\Delta u^2 > 2(\Delta u/2)^2$). This also includes changes of direction, given how they are accelerations in one direction and/or decelerations in another. These cases with high speed changes are, coincidentally, exactly the ones where the linear model differs the most from the nonlinear one, as seen in Subsection 2.5.4 and Fig. 2.19. This means that minimizing the slew rate not only makes

more theoretical sense given the nature of the variables at hand, but also helps improving the similarity between the linear model and the nonlinear one (and thus the real cloth too). This is of course because less changes in u result in a smoother movement, which is preferred, and if the reference trajectory does not demand it, sudden and constant changes will be avoided, which is especially useful in situations with noise, where the initial state might vary slightly on every iteration, and the resulting control signal with it if these constant changes were not penalized.

The new optimization problem formulation is shown on (2.21). As explained, in the objective function we now minimize the slew rates instead of the pure control signals, maintaining the weighting matrix R and the error part unchanged. The constraints are shown to make the changes introduced up until now explicit in one place. We can see the definition of the slew rates as a new constraint, which must hold also for $k = 0$, where we have $\Delta u(0) = u(0) - u(-1)$, using the local time indices where 0 is the current time step of the overall execution. This $u(-1)$ must be defined externally as an initial condition u_0, which is the applied control signal in the previous step. Before the initial state, we find a new constraint, introduced after the changes done in Subsection 2.5.3, forcing the distance between the upper corners (here compacted to d_{uc}, a 3-component vector, result of subtracting the corresponding states) to always remain constant and equal to the side length of the cloth, L. In the implementation, this equality is squared to remove a square root and keep a QCQP. The time instants are shifted like on the error computation, as we have no control over the current position at $k = 0$, but the constraint applies to the end of the horizon. The original control signals are still needed and bounded as before (maximum speeds are still relevant), so there is no need for a model or bounds change.

$$
\begin{aligned}
&\min_{u(k)} J = \sum_{k=0}^{H_p-1} \left[\left\| y(k+1) - r(k+1) \right\|_Q^2 + \left\| \Delta u(k) \right\|_R^2 \right] \\
&\text{s.t.} \\
&\left.\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) + f_{ct} \\
y(k) &= Cx(k) \\
\Delta u(k) &= u(k) - u(k-1) \\
\left\| d_{uc}(k+1) \right\|^2 &= L^2 \\
x(0) &= x_0 \\
u(-1) &= u_0 \\
x(k) &\in \mathbb{X} \subseteq \mathbb{R}^n \\
u(k) &\in \mathbb{U} \subseteq \mathbb{R}^m
\end{aligned}\right\} \quad \forall k \in [0, H_p - 1]
\end{aligned}
\tag{2.21}
$$

It is worth noting that in the code implementation, the slew rates are coded in a similar way to the state matrix. The optimization variables are still the absolute control signals, so a vector is created subtracting two consecutive ones (and the initial one in the first step), to be then used in the objective function, but internally, CasADi and IPOPT still have the same number of optimization variables. Of course, the previous control signal (in local coordinates) is also added to the input parameter matrix.

Changing the optimization problem like this is a considerable difference, which affects the tuning of all control parameters. Consequently, it also has an effect on computation times and tracking errors, which will be different for the new optimally tuned configuration. We must remember that after the MPC redesign introduced in Section 2.4, the computational time went down considerably, but was still around the 15 ms per iteration when $T_s = 10$ ms. As mentioned in that section, with the new times, altering T_s or H_p seem like reasonable strategies, and we have seen in Subsection 2.5.4 how altering T_s or the mesh size changes the parameters of the model, and how finding new ones is not such an easy task without Learning techniques, so we are left with an analysis of the effects of the prediction horizon.

With the baseline controller used with the linear model, a study of computational time and mean errors was carried out prior to this Thesis, resulting in the selection of $H_p = 30$, used also in the simulations shown in this document. For the redesign, this analysis will be shown here, for both versions of the objective function, minimizing pure control signals and slew rates, to check the effects of the change introduced in this subsection, and new suitable values of H_p for both versions.

The first step then is to obtain some well-tuned values for weighting matrices Q and R for both cases. In fact, Q is computed in an adaptive way as described previously, but we can call this matrix Q_a and add a constant weight Q_k. Now we can alter these two weights at will, knowing a higher Q_k will prioritize low errors, as they will be penalized more in the objective function, and a higher R will make minimizing the control signal term a priority. Beforehand, we found that $R = 20$ had the best results minimizing the control signals themselves (with $Q_k = 1$), but we can express these values normalized so the largest one is 1, as what is important is the proportion between them, not the value itself, and this way all weights will be between 0 and 1. With the same proportion, now $Q_k = 0.05$, $R = 1$. Another alternative is to make the weights add 1, leaving only one free value, which makes sense because the real degree of freedom is the proportion between two weights. However, with this method neither of the two weights is 1 (unless the other is 0), so the proportion itself is not as clear. It is also common to normalize the ranges of the variables being weighted, because sometimes the errors and the controls can be orders of magnitude apart. This is not the case in the considered optimization, though, because both errors and control signals are in the order of millimeters. With all these considerations, the resulting analysis for the new controller penalizing u can be seen in Fig. 2.22.

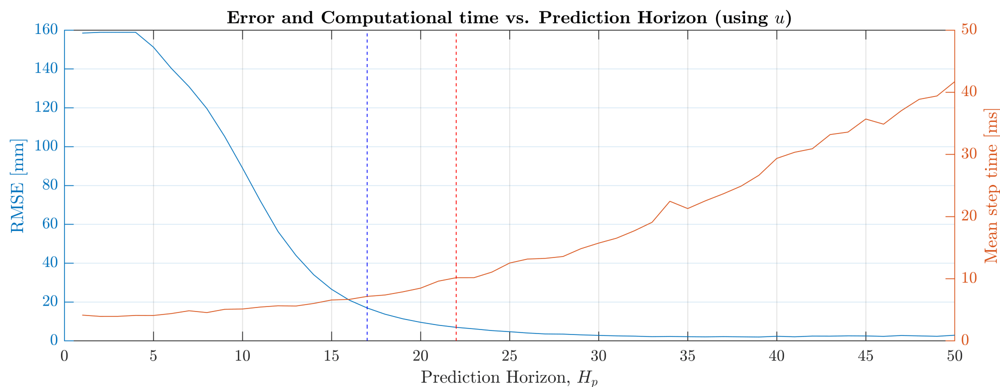

Figure 2.22: Analysis of the effects of H_p minimizing u with $Q_k = 0.05$, $R = 1$

In blue, corresponding to the left axis, the RMSE ("Norm" method) are shown, starting with large values for low H_p, which in fact represent simulations where the found optimal course of action is to stay in place. With such a short horizon, the tracking error is relatively low, and the minimal cost comes from reducing the energy consumption term, which is also weighted 20 times more. The errors decrease quickly between $H_p = 5$ and 15, and then they improve much more slowly. In fact, for $H_p = 17$, the RMSE is just a 10% higher than with $H_p = 50$. This percentage is an adequate threshold to divide low and high errors, and is marked with a blue dashed line.

Looking at the right axis, in red, we find the mean computational time per iteration, counting only the optimization and base changes, but not the SOM simulation. The evolution is the opposite of the error, as expected, taking considerably more time the larger the horizon is, as the optimizer must compute a prediction more steps into the future. The threshold value here is fixed by $T_s = 10$ ms, as optimizations must be completed within a time step. This value is crossed at $H_p = 22$, marked with a dashed red line.

Thankfully, a region to the right of the dashed blue line and to the left of the red one exists, giving us a range of values that result in fast and accurate enough simulations. This range is of course $H_p = [17, 22]$, and for example, choosing 20, we get a mean time around 8.9 ms with an RMSE of 11.5 mm.

We can now proceed to do the same analysis for the controller minimizing slew rates. Two different tunings were analyzed, with $Q_k = 0.05$ (Fig. 2.23) and $Q_k = 0.2$ (Fig. 2.24).

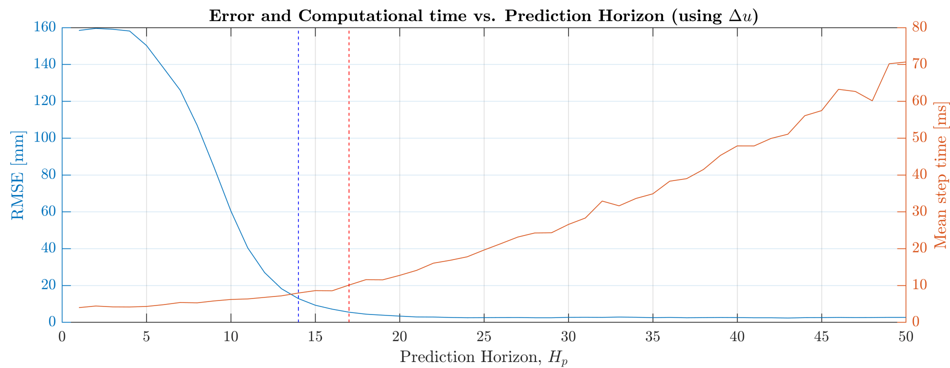

Figure 2.23: Analysis of the effects of H_p minimizing Δu with $Q_k = 0.05$, $R = 1$

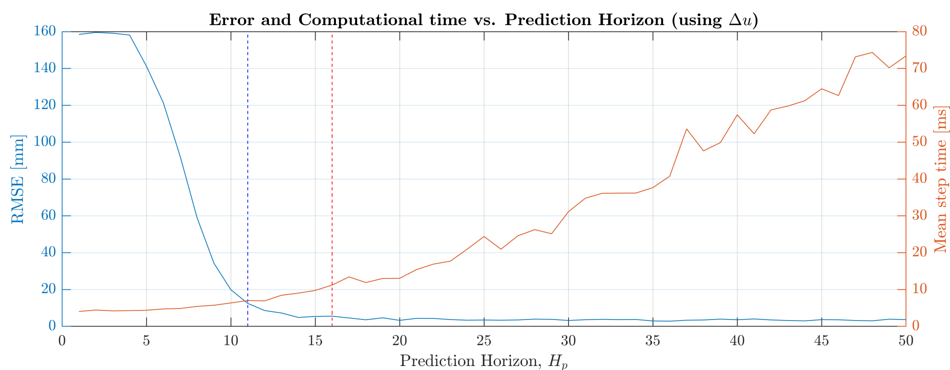

Figure 2.24: Analysis of the effects of H_p minimizing Δu with $Q_k = 0.2$, $R = 1$

Figures 2.23 and 2.24 are left close together for a better visual comparison. The first tuning is the same one as in the case minimizing u. The overall behavior is the same, as one would expect but the errors decrease faster, reaching the 10% threshold at $H_p = 14$. This improvement is countered by the fact that mean times increase with lower horizons, crossing 10 ms at just $H_p = 17$, leaving a very thin range of possibilities. Simulating with the new controller minimizing Δu to find better results using other weights, an optimal tuning was found at $Q_k = 0.2,\ R = 1$. However, even with a horizon of 20 steps, this resulted in an average computational time of 13.6 ms per step and a RMSE around 3.3 mm. Performing the same analysis as before with the new weights, we can see how the errors descend much faster reaching the 10% threshold at just $H_p = 11$. The times seem to increase ever so slightly, with the 10 ms mark being crossed at $H_p = 16$ now. This leaves a reasonable range of possibilities for the horizon to obtain quick and accurate simulations, even if with such low values of H_p and fast sampling time, the total prediction time goes down from the original 0.3 s to around half of that, at 0.15 s if we choose $H_p = 15$.

These analyses show several important points to consider. Firstly, tuning is key and influential on errors and times. Secondly, the proposed redesign, even minimizing pure control signals, can reach mean iteration times under $T_s = 10$ ms while only reducing the horizon 10 steps, barely increasing the tracking error. Using the slew rates can prove a bit more resource intensive, needing even lower horizons, but this is compensated by the tracking errors being noticeably lower too. All in all, with the applied changes, an implementation in real time now seems completely feasible, even before increasing T_s. Nonetheless, total prediction times have been considerably reduced from an already small starting point, so increasing the time step (after finding the new parameters) will still be useful to increase this value and take full advantage of a predictive controller.

2.5.6 Simulating Closer to Reality

All the major and minor changes until now have been introduced to be able to convert the control scheme implemented in simulation into a real system. This means generalizing what initially was a very specific case or adapting to the needs of a real case. However, in all the executed simulations, the ideal case was considered, where all the data was known and used by all the connected parts of the control scheme with no issue. In a real implementation, even with a sensor like a camera that provides the full state vector, we will have noise, disturbances, and other unexpected behaviors. This section is dedicated to preparing for these situations, even if we are still in simulation.

First of all, the main problems we can find when dealing with real signals between different components are disturbances and noise. As we saw in Fig. 2.2 and in (2.4) for a linear system, disturbances $d(k)$ affect the difference equation through B_d, and thus the evolution of the states themselves, while noise $n(k)$ is only found in the output signal. While the referenced equations are for a linear case expressed in state space, this distinction holds in general, also for nonlinear systems. To give an example related to the studied case, a disturbance could be the force of the wind or someone touching the cloth piece during the execution of the task, unforeseen interactions that change the actual state of the system in ways not predicted by the controller. Noise comes from the sensors, and could be the uncertainty or maximum precision a camera and its computer vision algorithms produce.

Adding these signals to the simulations is quite straightforward if we consider they follow a random normal distribution instead of a more complex model. For example, we can create a disturbance vector with three components, each one following a Normal distribution ($d_i(k) \sim N(0, \sigma_d)$), corresponding to an unexpected movement in each one of the coordinate axes, so that all nodes move synchronously. Then, even in the nonlinear model, we can have a matrix $B_d \in \mathbb{R}^{6N \times 3}$ that relates these three values to all the states, which are positions and velocities of all the nodes in all three coordinates, hence the size $6N$. Instead of it being a direct change in position, we can consider the disturbances as a change in velocity and let the system evolve accordingly, leaving the first $3N$ rows of B_d as zeroes. The rows corresponding to the velocity of the controlled nodes must also be set to 0, as they are being grabbed by a robot (or connected to the rigid piece with only one arm for both corners), and a slight disturbance in the cloth will not be enough to move the entire manipulator too. The noise can be another random signal following a Normal distribution ($n_i(k) \sim N(0, \sigma_n)$), however, in this case it is not coming from a physical action on the system, so the states will not move following any logic, it is just added sensor noise, and we can generate one independent value per state. Given a camera would sense the positions and compute the velocities from there, we can just add noise to the first half of the state vector, which would be the output vector $y(k)$ in this situation.

Once we have random variables in our system, the next logical step is to analyze the options to consider them also in the controller, to try and work with them and take profit from the fact that we know they will be present. That is changing from a Deterministic approach of MPC to Stochastic MPC (SMPC), in any of its forms, the most common ones being a Naive approach, Chance-Constrained MPC (CC-MPC), Tree-Based MPC (TB-MPC) and Multiple-Scenario-Based MPC (MSB-MPC) [26], [27], [28], [29]. Without going into the details for all of them, it can be implied from their definition, and is also proven in the literature that compares them, that both TB-MPC and MSB-MPC are slow approaches, involving forecasts of multiple possible futures considering different disturbance values. The only two implementations to be considered in our application are a Naive approach and CC-MPC. Between the two, CC-MPC requires statistical knowledge of the uncertainties, and a reformulation of the control problem involving a risk of violating the constraints due to the disturbances, while the Naive approach is much simpler (and faster), with the only change being the addition of an estimated disturbance to the COM equations. With this in mind, and knowing computational time is still our main limiting factor, the Naive approach will be implemented, so we can check its effects and proceed accordingly.

With the Naive approach, the objective function on the optimization problem is unchanged, and only the model constraints are different from before, now including a prediction or estimation of the disturbance ($\hat{d}$), which must be given as an initial parameter if we have a closed "Solver" object, for them to be different on every call. We can see this in (2.22).

$$
\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) + f_{ct} + B_d d(k) \\
x(0) &= x_0 \\
d(k) &= \hat{d}(k)
\end{aligned}
\tag{2.22}
$$

ETSEIB

This estimation of the disturbance can be done by generating random values with the same distribution different from the ones really applied to the SOM in simulation. This requires knowing σ_d, or at least have an accurate approximation for it. In simulation, we can just use the same number and assume we could obtain it analyzing a real scenario. For the tested executions, a disturbance of $\sigma_d = 0.003$ m/s was added. This corresponds to an added velocity between 0 and 9 mm/s ($3\sigma_d$) in more than 99% of the cases, which is a considerable value knowing the input displacements are around 0.5 mm/step which would be a speed of 50 mm/s. Sensor noise is also added at $\sigma_n = 0.001$ m, which also seems a reasonable amount.

Under these conditions, a total of 10 simulations were executed with uncertainties but still using a Deterministic MPC (DMPC) approach, and another 10 simulations were done using the Naive SMPC approach. In fact, before the 20 simulations done to analyze the results, several others were done to tune again the objective function weights, as adding the new random signals changed which values resulted in the best tracking and errors. The prediction horizon was fixed at $H_p = 15$, following the results obtained in Subsection 2.5.5, as now we are minimizing slew rates. The new tuned weights are $Q_k = 0.01$, $R = 1$. The quantitative results of the analysis are shown in Table 2.3, where the obtained means and standard deviations of the 10 samples for each type of MPC are shown for both the RMSE ("e" subscript) and the computational times ("t" subscript). We have used μ and σ to avoid confusion, but it is important to note these are sample means and deviations, usually referred to as $\bar{x}$ and s.

Table 2.3: Quantitative results of DMPC and SMPC simulations

Case	μ_e **[mm]**	σ_e **[mm]**	μ_t **[ms]**	σ_e **[ms]**
DMPC	14.3100	0.6501	9.7616	0.2047
SMPC	13.7872	0.5274	9.8516	0.2521

We can see how the mean error is indeed lower when we use SMPC, even with the implemented Naive approach. The computational times increase marginally as a consequence, with multiple simulations having values over 10 ms, but the average remains under this value. However, these slight differences might a result of the specific samples, and the distributions might not even be statistically different. As a first step, we can plot them to have a visual representation of the results, seen in Fig. 2.25.

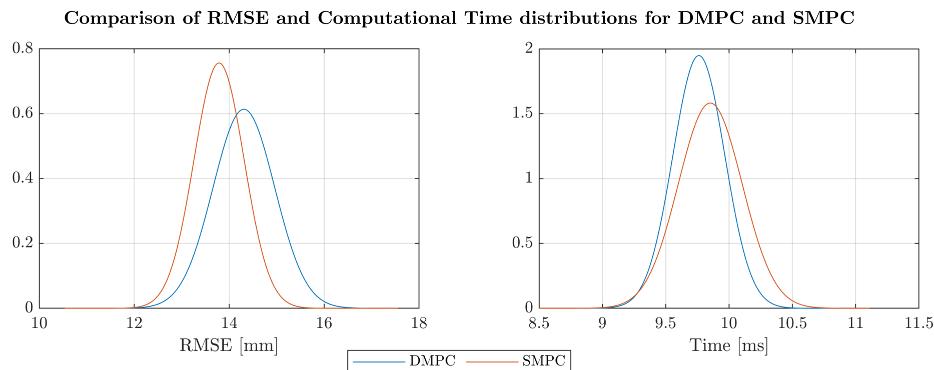

Figure 2.25: Resulting distributions for the results of the DMPC and SMPC simulations

With just this plot, we can already see how this distributions are really similar, especially the ones comparing time. In fact, to justify the implementation of SMPC, we would want a notable improvement (statistical difference) in the errors without significant changes in computational times. We can numerically check if two Normal distributions obtained with samples are statistically different with a Z-test [30], applying the formula shown in (2.23), where we have used $\bar{X}$ and s to be rigorous, but these are of course the previously obtained sample μ and σ.

$$Z = \frac{\left|\bar{X}_i - \bar{X}_j\right|}{\sqrt{\frac{s_i^2}{N_i} + \frac{s_j^2}{N_j}}} \tag{2.23}$$

We obtain $Z_e = 1.9749$ for the errors and $Z_t = 0.8755$ for the times. This means there is no statistical difference between the time distributions, which means the negative effect of SMPC is not significative, but the value for the errors is on the border of being “marginally different”, so the improvement is also not proven significant. Considering the controller must be implemented into a real system, that the models of the disturbances there are unknown, and that the computational times already are at the feasibility limit, we will discard the SMPC option for the first real implementation, but have this analysis always present as a possible future improvement.

As a final note regarding SMPC, in Fig. 2.26 one of its simulations is shown. We can clearly see how the noise and disturbance affect the lower corners greatly, in the Y-axis in particular. However, the upper corners have smoother evolutions, both as a result of them being picked by the robot and because we are minimizing the slew rates (Δu), so a nervous evolution would give a high cost.

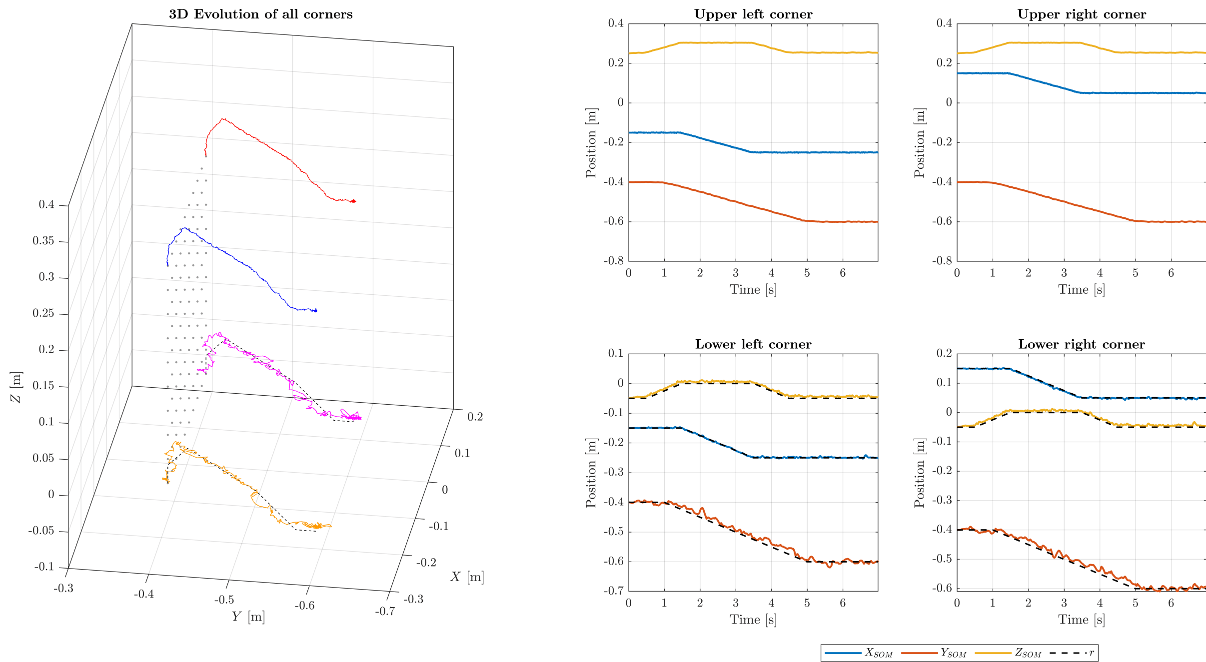

Figure 2.26: Results for a simulation using SMPC

Independently from SMPC, while trying to create simulations as close to a real case as possible, a strange behavior was detected on the first iterations of the nonlinear model. If the cloth was initialized in a completely vertical plane, with one of the coordinates being the same for all the nodes (for example, if the cloth was initialized in the XZ plane, all the nodes have the same Y component), the first iterations were not reliable until the cloth reached settled in a steady state. This caused warnings on the simulation, and also some oscillations on the resulting closed-loop evolution, as the controller received incorrect initial states. To solve this issue, before starting the trajectory tracking loop of the simulation, a smaller loop was added in which the nonlinear model is simulated until it behaves correctly, when it settles and the warnings disappear. The simulations done to analyze the SMPC were done with this already in place, as the behavior was detected in time and the results would not be accurate otherwise.

Continuing with the objective of simulating closer to a real scenario, and when the real implementation had some progress to it, as briefly mentioned in Subsection 2.5.4, it was clear that the slowest part of the system was the Vision feedback (detailed in Section 4.4), taking several time steps. This was the justification behind a simulation code using a linear SOM, to check its evolution and then, instead of substituting the SOM for the real system as it would be usual, keep both in the real implementation, with the linear SOM as a "backup" to simulate the evolution of the cloth while there is no new real feedback information. However, the simulation using a linear SOM and the final implementation differ greatly, and a design change of this caliber is better tested first on simulation, representing correctly all of its parts.

This meant a redesign of the simulated control scheme to match the real triple model scenario that will happen on the final implementation, with the MPC block left untouched, a linear "backup" SOM, and adding the nonlinear model to represent the real system. With this new scheme, shown in Fig. 2.27, the linear SOM will be simulated on every time step to obtain updated initial states to call the optimizer with, so it can compute correct control signals with current data, even if it is an approximation between real feedback signals. These control signals will be sent back to the SOM to close this fast simulation loop, and also to the nonlinear model (real cloth). When new feedback data is received, its mesh is reduced and processed, and the states of the linear SOM are updated accordingly (immediately updating x_0 too).

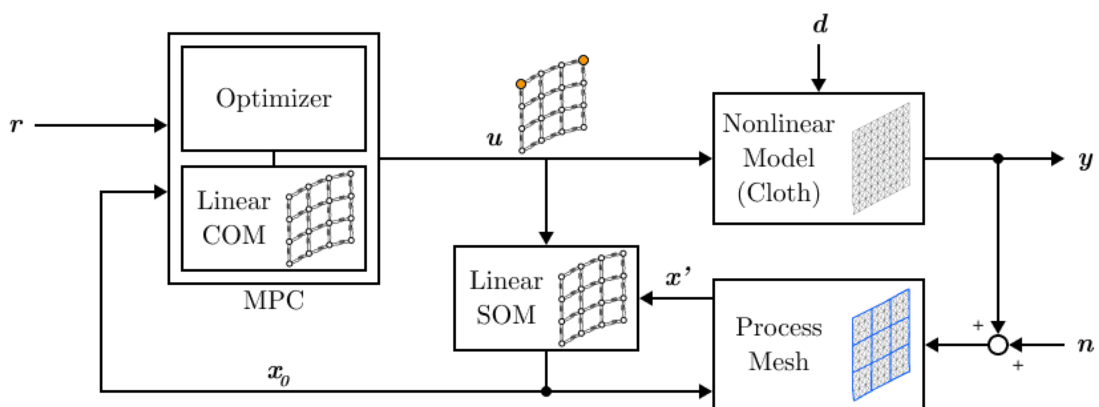

Figure 2.27: Closed-loop block diagram of MPC applied to cloth manipulation in simulation

The full code implementing this triple model scheme can be found in the Appendices document. The feedback processing methodology and details about closing the loop using real feedback can be seen in Subsection 4.4.3, where we also explain why there is an arrow from the linear SOM to the Process Mesh block, and how a reliable SOM can positively impact the final control scheme under heavy noise, disturbances, or lack of feedback for prolonged periods.

Finally, as a final step to bridge the gap between simulation and reality, and to check that the computed TCP poses are reachable by the real robot, its model was also added to the simulation, or, more precisely, to all three closed-loop simulation codes, with a nonlinear SOM, with a linear SOM, and the new triple model one. At this point in the Thesis it was clear that the real implementation was going to be using a Whole Arm Manipulator (WAM) robot from Barrett Advanced Robotics, available at the Perception and Manipulation laboratory in the IRI. Knowing which robot is going to be used and its specifications is very useful to determine if a TCP pose obtained from the positions of the upper corners (output of the controller) is reachable, and what will be the configuration of the robot in case it is. In fact, this process of converting from a pose (position and orientation) in Cartesian space to a configuration of the robot (the position of its joints, or joint space) is what is known as Inverse Kinematics (IK), and what a Cartesian controller does. This is why this whole process is explained further in Section 4.3, where the parameters of the used WAM are also shown. What is relevant for the simulations is that using these parameters and the Robotics Toolbox by P. Corke [31], we can compute the configuration of the robot, and plot its model too. We can give it all the poses of a simulation, and if one is not reachable, an error will pop, indicating the reference trajectory must probably be changed so the resulting TCP one is inside the workspace of the robot. We can also use this change to create plots with the WAM, such as Fig. 2.28.

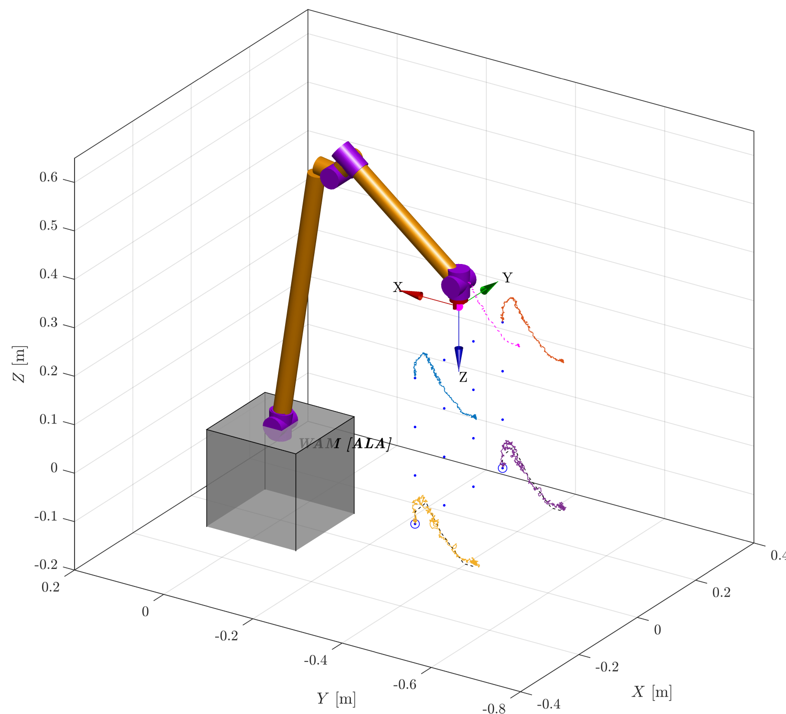

Figure 2.28: 3D plot with the results of a simulation, including the WAM Robot

2.5.7 Flexibility Options and Summary

This final subsection summarizes the changes done throughout the chapter, and also shows some tweaks added to enable or disable several functionalities, or choose from different alternatives.

First of all, the baseline controller was redesigned from scratch, keeping only the same cloth models as COM and SOM, but reducing the complexity of the optimization problem, resulting in faster executions (from 50 to 15 ms per step, using $T_s = 10$ ms) without increasing the tracking errors.

This redesigned MPC was then modified to accept linear models (meshes) of any size. To be able to introduce rotations in the reference trajectories, a local cloth reference frame was introduced to simulate with the parameters in the correct local axes. Given a singular robot is going to be used in the real implementation, the pose of its TCP is also computed as an output.

New simulations were also created using different models: an updated nonlinear SOM, a linear model also as SOM, and a triple-model scheme, with an additional "backup" linear SOM, necessary to simulate the system and update the initial conditions of the optimization problem when there is no real feedback data. This scheme is what the real implementation requires, as the Computer Vision algorithms used can take several steps of the controller (around 100 ms) to output the most recently captured data.

The optimization problem was changed to penalize the slew rates (Δu) instead of the pure control signals (u), and a Naive Stochastic MPC approach was considered to reduce the effects of noise, but it was discarded as the results were not significantly better. Switches were added to choose between:

- Minimizing the control signals (u) or slew rates (Δu) in the objective function.
- Using the adaptive computation of the weighting matrix Q_a together with the constant weight, so $Q = Q_a Q_k$, or disabling the adaptive part and using only the constant weights.
- Considering an estimation of the disturbance in the COM to implement the Naive SMPC approach, or not, simulating regular DMPC.
- Showing an animation of the cloth mesh performing the resulting trajectory or just a static 3D plot of the initial situation and the corner trajectories. If the animation is enabled, a model of the WAM robot can optionally be shown.

The first two options affect the structure of the predictive controller, and will be used to find the most suitable one in Section 3.3. Finally, the parameters of the linear model (or models, depending on the simulation executed) are loaded from an external table. After completing the Learning process described on Section 3.2, this table will contain a set of parameters for each of the studied mesh sizes and sampling times. With just these two values, the correct combination can be loaded.

This concludes all the changes introduced to the MPC, to have more versatile and realistic simulations, while also improving its performance. As mentioned throughout this chapter, the final closed-loop simulation codes can be found in the Appendices document.

3. Reinforcement Learning Enhancements

The present chapter is centered around the applications of Reinforcement Learning (RL) used in this Thesis to improve the newly redesigned Model Predictive Controller even further. Section 3.1 introduces the topic of RL, takes a look at the state of the art, and explains some theoretical concepts needed to understand the specific methods and algorithms used. Section 3.2 describes the first application of RL in the project, to find the optimal parameters of the linear cloth model so it behaves like the real cloth, while Section 3.3 shows how RL can be applied to tune the parameters of the controller itself.

3.1 Theoretical Background

Reinforcement Learning (RL) is a wide and constantly evolving topic of research under the even larger umbrellas of Machine Learning (ML) and Artificial Intelligence (AI). As such, it revolves around the idea of creating an algorithm capable of understanding data and acting in consequence, without being explicitly programmed. Specifically, RL focuses on obtaining these situation-to-action relations, or more simply, learning how to behave, through trial and error interactions with the environment [32].

The common diagram for a RL scenario is shown in Fig. 3.1. In it, we find an agent, which is the entity that must learn, with a policy and the RL algorithm itself inside. The policy is what will be learnt, a decision (or sequence of them) that must be made depending on the current situation, or state. Observing the environment, the agent will receive this state, and act according to its policy. The action will produce a change in the environment, from where we can observe the next state and also extract a reward, measuring the performance of the action taking. All this information (action, previous and next states, reward and policy) is taken by the RL algorithm, which also saves previous data, and will keep updating the policy in order to maximize the obtained rewards.

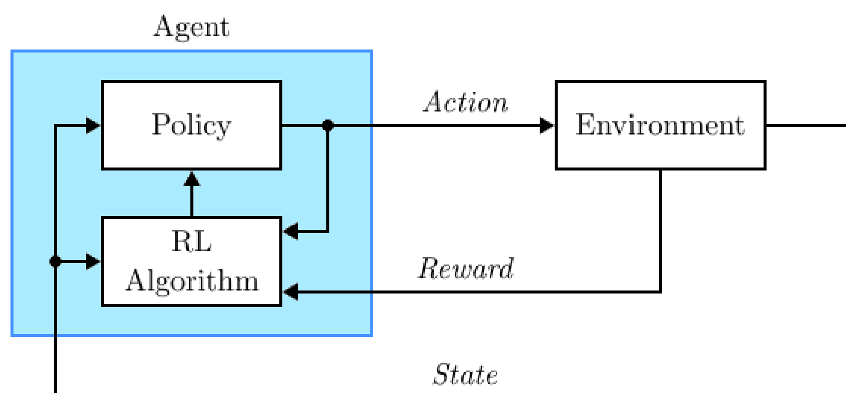

Figure 3.1: Basic diagram of Reinforcement Learning

ETSEIB

In the shown diagram, the reward is extracted from the environment directly, but another common way of representing this situation is to have an Actor-Critic structure. The actor would be the agent of the shown diagram, and the critic evaluates the taken action giving a reward depending on the evolution of the states. These names give a pretty visual image of someone acting according to what they have rehearsed, while receiving performance reviews from someone else, and improving thanks to this feedback.

Even if this is the common situation for RL, this topic alone covers a wide variety of problems and different algorithms to solve them, considering deterministic models where taking a determined action on a given state will always result in the same evolution, stochastic ones where there is a chance to have a different evolution even with the same conditions and actions, model-free approaches, situations where the reward depends on just the present or considers possible future events, and the list goes on with several approaches to each specific situation too [33], [34].

Looking at the applications of RL in the Robotics field, this variety does not diminish. In the recent years, research has started focusing on robots not only on enclosed industrial environments, but also in everyday situations, where the surroundings can be more varied, the variety of tasks is bigger, and, usually, there are humans nearby, as assistive robotic applications have seen a radical increase too. In these applications, RL is usually found to learn how to perform an arbitrary task, sometimes assisted by Learning from Demonstration (LfD), where the task is first done by a human expert so the robot has a starting point and some data on successful outcomes [35], [36], and RL can be applied to find policies that reach performance levels even beyond those of the human expert.

A prominent approach to RL in Robotics are the policy search algorithms. In them, a given policy is parametrized and the focus is learning the parameters that produce the optimal outcome [37]. However, in policy gradient approaches, information is lost upon updating and improving the policy, and they are either heavily influenced by starting conditions, usually optimizing to a local minimum without exploring further, or produce infeasible solutions. This motivated the creation of a new algorithm, named Relative Entropy Policy Search (REPS), which estimates new policies based on data distributions, bounding the information loss [38]. Research at the IRI has used this method successfully in robotic applications, for example parametrizing movements with Probabilistic Movement Primitives (ProMPs), which capture the variance of a set of demonstrations [39], and even improving the method with the proposed Dual form (DREPS) seen in [40].

During Chapter 2, the need for a learning approach to find the parameters of the used cloth linear model was made evident, especially in Subsection 2.5.4, where their dependency on mesh size and sampling time is shown, and using a general optimization failed, due to all the mesh nodes through the entire trajectory depending on the parameters of the model, so only reduced meshes and trajectories worked without reaching computational limits, yielding poor results. It is clear how having a trajectory depending on some parameters is the same case as the base of Policy Search, parametrizing a policy. Among these methods, REPS was chosen as the most suitable one, given the proven results in situations similar to the one studied in this Thesis, and the experience gained from that research at the IRI itself.

Once delving into the world of RL, we can think beyond the previous application and try to improve the Model Predictive Controller itself with learning techniques. In several parts of Chapter 2 (e.g., when redesigning the controller, when using other models, or when minimizing the slew rate), it was clear how tuning the MPC was key to improve the performance, and the conditions also affect this tuning, which had to be performed manually until the best outcome is found. Furthermore, we were limiting the weighting matrices (with size 6×6) to have the same values for all components, when they could be different for each one. This amount of parameters and their effect on the control problem justify also applying RL to better tune the MPC.

Even if their fields are independent and they have evolved separately, RL and MPC share some similarities, which have already been reported in literature [41]. Mainly, both feature an optimization problem at their base. In fact, a much greater similarity can be found between RL and Adaptive Control, which also tries to find the value of some parameters to improve the control inputs applied based on past data [34]. Adaptive Control is focused on estimating the model from input and output data, and updating the controller with progressively better estimations. This is a more mature field, using parameter estimation techniques on models with fixed structures, but even then, it has its shortcomings, as it works together in two opposing ways: good estimations need rich data, with varied values, to have more information about the input-output dynamics of the system and how it behaves, so an estimator needs a constantly moving output signal, but a controller wants exactly the opposite, usually wanting to lead the system to a steady state. This problem is reduced with an accurate initial identification of the model using rich data before closing the loop, and this type of control is widely implemented, with one particular case being of course Adaptive MPC, where new data helps tune the COM [42]. Changing this scheme to one using RL is not such a big leap, only needing to swap the method to obtain the improved model. However, this learning is also focused on the model, not tuning the controller.

Other interesting combinations of RL and MPC can be found in the literature. It is common to see structures like the one found in [43], where there are several control layers, with a direct feedback control, an MPC on top, and then a learning layer. The application shown in [44] is very notable, as it actually uses an Actor-Critic algorithm to learn the objective function of the MPC itself (and the Related Work section is very illustrative too). This idea is much closer to the mentioned objective for our application, letting the RL handle the MPC tuning.

With this last approach, we are once again in a situation where Policy Search can be applied. The parametrization of the policy here is a bit more complex, as the parameters are found in the objective function of the MPC, but the concept is the same, changing the parameters changes the outcome of the policy (if the optimization weights change, the control signals will be different). This affects the state of the environment, and a reward can be computed accordingly, for example, corresponding to tracking performance or also accounting for computational time. Given we are in the same type of situation as before, the chosen learning algorithm will be REPS too, for the same reasons as before.

With the state of the art analyzed and the algorithm to be used chosen (both considered applications will use REPS), what remains of this section will be dedicated to explaining the basics of this method.

The entirety of the algorithm will not be replicated here, as it can be found in [38], but some remarks can be added in combination with [40] to adapt it to our described applications, where we want to find the parameters that describe the optimal policy, be it the model parameters so its behavior is similar to the real cloth or the objective function weights to obtain the best tracking.

With this reasoning in mind, a policy $\pi(\theta)$ is represented by a multivariate Normal distribution, with a vector of means μ and covariance matrix Σ. The samples generated can be written as $\theta \sim N(\mu, \Sigma)$. In the REPS algorithm, the Kullback-Liebler (KL) divergence is used. This indicates the difference between two probability distributions (p, q) over a random variable (x), as seen in (3.1).

$$\mathrm{KL}(p||q) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{3.1}$$

This indicator is used in REPS to give a divergence bound ϵ to the difference between a given previous policy, $q(\theta)$, and a resulting one, $\pi(\theta)$. With known associated rewards $R(\theta)$, the optimal policy π^* is found as the solution of the optimization shown in (3.2), which is proven to be (3.3) in the literature.

$$\begin{aligned} \pi^*(\theta) &= \arg\max_{\pi} \int \pi(\theta) R(\theta) d\theta \\ \text{s.t.} & \\ \epsilon &\geq \mathrm{KL}(\pi(\theta)||q(\theta)) \\ 1 &= \int \pi(\theta) d\theta \end{aligned} \tag{3.2}$$

$$\pi^*(\theta) \propto q(\theta) \exp\left(\frac{R(\theta)}{\eta}\right) \tag{3.3}$$

When performing policy iteration with several samples, the exponential term of the solution acts as a weight for all the samples $q(\theta_k)$ of the previous policy, in order to obtain the new policy considering the corresponding reward for each one ($R(\theta_k)$, or r_k). The new value η is the Lagrange multiplier for the KL bound, obtained by optimizing (3.4), the dual function of the problem in (3.2).

$$h(\eta) = \eta\epsilon + \eta \log\left[\frac{1}{N}\sum_{k=1}^{N} \exp\left(\frac{r_k}{\eta}\right)\right] \tag{3.4}$$

To put all these concepts together into an algorithm we can use to obtain the optimal parameters, we show the final step by step process in Algorithm 2. In fact, the application of d_k mentioned in line 8 corresponds to (3.3), and this weighting can be done with a simple weighted average of the executed samples, or using Weighted Maximum Likelihood Estimation, for example. In the update, we obtain the new means and covariance matrix of the parameters that describe the policy, so after several policy updates (epochs), we have the optimal parameters, which is what we are actually looking for.

Algorithm 2 Policy iteration using REPS

Require: $N, \epsilon, q(\theta)$

1: **for each** policy update or epoch i **do**
2:     **for** $k = 1 \cdots N$ **do**
3:         Perform an experiment with θ_k, a sample from current policy $q(\theta_k)$
4:         Compute reward r_k
5:     **end for**
6:     Optimize the dual function h to find η
7:     Find relative weights $d_k = \exp\left(\frac{r_k}{\eta}\right)$
8:     Apply d_k to the corresponding $q(\theta_k)$ to find the new policy $\pi(\theta)$
9:     $q(\theta) \leftarrow \pi(\theta)$
10: **end for**

In case completing the experiment for each sample is slow, and the number of samples or rollouts per epoch N is low in consequence to have learning processes over reasonable amounts of time, we can help this process by considering experiments from past epochs, and computing relative weights for $2N$ samples and rewards. This will be especially useful in Section 3.3, where every experiment is a closed-loop simulation with the MPC controller in action.

The considered experiments are very sensitive to the parameters they depend on. When learning the parameters of the linear model specifically (Section 3.2), even a slight increase in stiffness can result in the model being unstable, and the rewards can be several orders of magnitude lower, even being considered as $-\infty$ or Not a Number (NaN) by the program executing the algorithm. Even filtering these extreme cases, having rewards that differ so much makes the relative weights d_k basically be either 0 or 1 (simply indicating "went wrong" or "nice"), which removes the differences between the successful, stable experiments that would make the policy update move towards optimal values. This can be fixed by adding a minimum reward limit, where results under this value saturate to it, indicating any reward under this value should be considered as equally wrong. The value itself must be low enough so the relative weights of the corresponding experiments are almost if not exactly 0, but not too low so the rest of them lose their proportional differences, and the new means can evolve towards the best values among them, not only those that worked. This modification can accelerate the learning process greatly, and reduce the cases where, even after several updates, the number of successful (stable) samples is low. Even then, the first epoch might result in all the samples being unstable given some initial distribution, in which case the policy is not updated and the epoch is repeated.

Another slight modification done to the basic REPS algorithm is to add a factor λ to the diagonal of the covariance matrix Σ, in order to increase exploration a bit (making it more likely to choose sample parameters a bit further away from the mean) and reduce the chance of falling in a local optimum and never leaving it. This factor can be obtained, for example, as proportional to the mean of the Singular Values of Σ, so it is larger on the first epochs, and decreases as learning progresses, and the variance goes down to have samples closer and closer to the optimal policy.

With the used algorithm and the performed modifications explained, we can now focus on the work done for each one of the two applications of RL developed in this Thesis.

3.2 Learning the Parameters of the Linear Model

The first implementation of RL was to find the parameters of the linear cloth model shown in Section 2.3 when changing mesh size and/or the time step (T_s). Expressed in a local base, there are a total of seven parameters to be found, stiffness and damping for each coordinate plus the super-elastic correction parameter, Δl_{0z}. The parameters of the original model, a mesh of size 4×4 and discretized with $T_s = 0.01$ s, were found through optimization, minimizing the difference between its evolution and that of a nonlinear model. In fact, only the difference between the positions of the lower corners was considered, and not the rest of the mesh nor the velocities. This was possible given its small proportions, and using a relatively short input trajectory, as the optimizer could not reach a feasible solution in other situations (even increasing the maximum allowed time, the resulting parameters produced unstable evolutions).

Following this idea, the first implementations of REPS to find the linear model parameters were also done comparing its evolution to the nonlinear model, or more specifically both versions used, explained in Section 2.3 and Subsection 2.5.4. The obtained rewards, however, depended on the distance in position from the complete linear mesh, instead of just the lower corners.

It is important to note that these comparisons are open-loop simulations, simply registering input-output data of the models. The input is a trajectory for the upper corners of the cloth, computed arbitrarily beforehand, and applied to both models at the same time to save their respective evolutions and obtain the differences between them. This is mentioned now because this is also much easier to implement in a real scenario than the full closed-loop scheme, and thus data from the real cloth could be gathered quite early on in the duration of this Thesis, with just a Cartesian controller, the robot and a Vision algorithm, and no need to connect the MPC. Even the processing of the Vision data could be done offline after the experiments. Having access to real data meant being able to completely skip the nonlinear model, as it was a step between the linear one and the real cloth, and make the linear model, which is the one used in the controller, behave directly like the real cloth, given the same input trajectories.

With this in mind, the rest of this section is divided in the following sequential steps to obtain the optimal model parameters for different sizes and sampling times, using only data obtained from the real cloth, and setting aside the previous experiments done with the nonlinear model, as it is now an unnecessary step between what will actually be used in a real implementation (even adding the "backup" SOM mentioned through Section 2.5 due to the feedback being slow, this model would be linear to be computationally fast). Each one of these steps is explained in its own subsection: first, the creation of the input trajectories and offline processing of the gathered real data in Subsection 3.2.1, then the learning process itself using this data in Subsection 3.2.2, and finally an analysis of the different learning experiments, and the definitive linear model parameters in Subsection 3.2.3.

3.2.1 Gathering and Processing Real Data

To begin the data gathering process, first we need to create a dataset of input trajectories that are executable by the real robot. With the modification introduced at the end of Subsection 2.5.6, once knowing the real robot was going to be a WAM by Barrett, the feasibility could be checked in simulation. Some trajectories, seven to be precise, were already created for the experiments performed with the nonlinear model, but they either had some unreachable points or were situated in a region of the workspace that had obstacles in the real scenario at the IRI laboratory.

Some of these trajectories were variations on the one used in the optimization to obtain the original parameters, making it longer, faster and/or with more resolution. To keep this trend, trajectory 8 is simply the final one of the previous set moved to a feasible and free region of the workspace. Trajectory 9 follows the same trajectory but adds a final fast rise at the end. Trajectory 10 is radically different, with a semicircular motion contained in the YZ plane, and trajectory 11 has a fast diagonal movement and then a circular motion (quarter of a circle) in the XY plane. We have kept the original numbering instead of starting from 1 for the ones shown in this document in case a reader or future researcher wants to check the codes attached or linked in the Appendices, so they can see direct correspondence with the files. These four trajectories are shown in Fig. 3.2, where the WAM model is included for the first one.

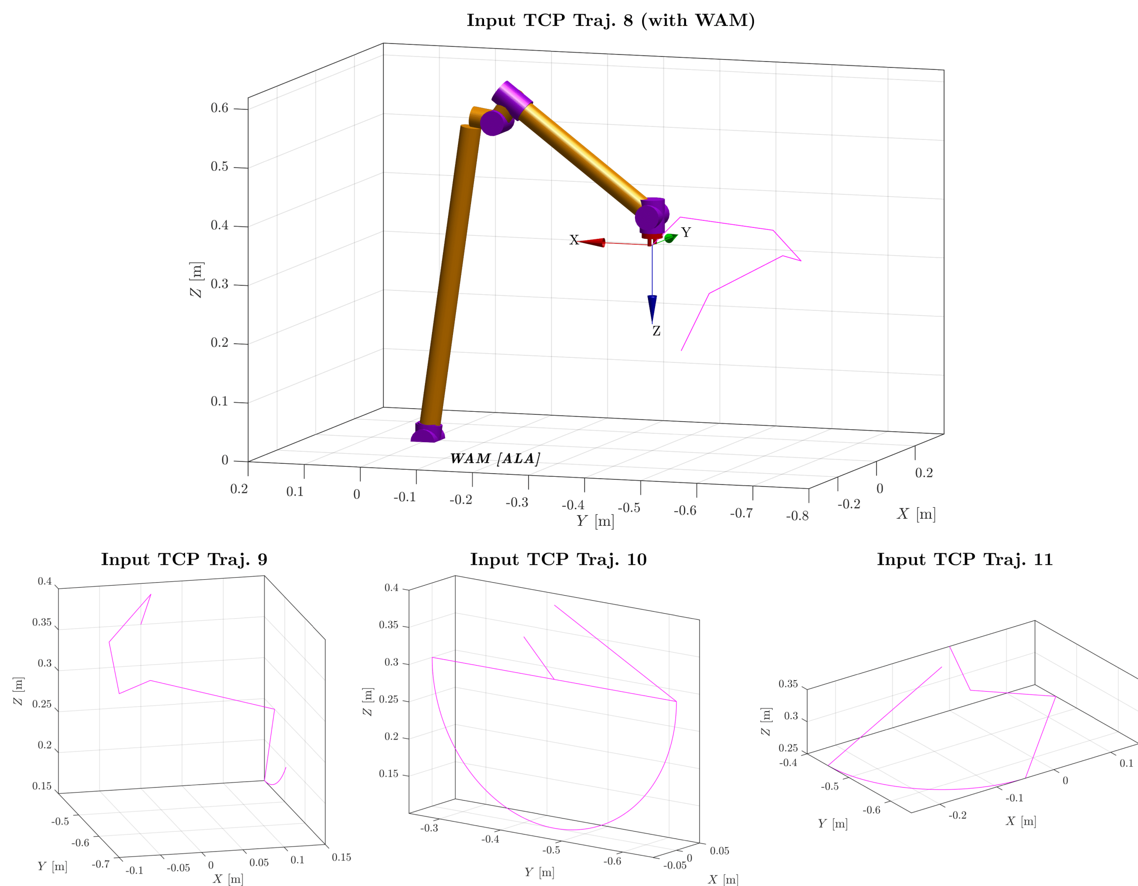

Figure 3.2: 3D plots of all the used input TCP trajectories

In theory, if we want versatile model parameters that can be used with any trajectory, we could learn them executing always the same one, including movements in all possible directions. However, if the learnt parameters are significantly different depending on the situation, it might be better to apply a combination for each specific case, learning which are these cases too, instead of a one-fits-all set.

These input trajectories are for the TCP, which is higher (offset Δh) than the upper corners of the cloth. They also contain a full pose for each point, not only a position. It is important to note that even the circular motions are still translations, and the orientation of the TCP is always constant, with Z^{TCP} pointing vertically down like shown in the figure. This is because the parameters depend on the orientation of the cloth, and will be applied to its local base (for example, as seen in Subsection 2.5.2, stiffness in the direction perpendicular to the cloth plane is much lower, and this property must be kept), so it is easier to keep one orientation for the whole learning process, learn the parameters easily when the change of base is the identity ($R^{cloth} = I$), and apply rotations afterwards. In fact, two additional trajectories, 12 and 13, were created including rotations, but were used to validate the behavior of the system in this situation, and not to learn.

The details on the method used to execute the trajectories can be seen in Section 4.3, concretely the "Read Node" explained there. The process to obtain output data through Vision is explained in detail on Section 4.4, including how to calibrate and change from camera coordinates to robot or world ones. Even if for the first sessions some of the processing code was not ready and running online, so the gathered data was raw, in coordinates relative to the camera, the processing done offline is exactly the same, so we refer to that section to avoid repeating the same processes. However, this data (Fig. 3.3 shows an example) must go through some extra steps to be used for learning, and those will of course be explained here.

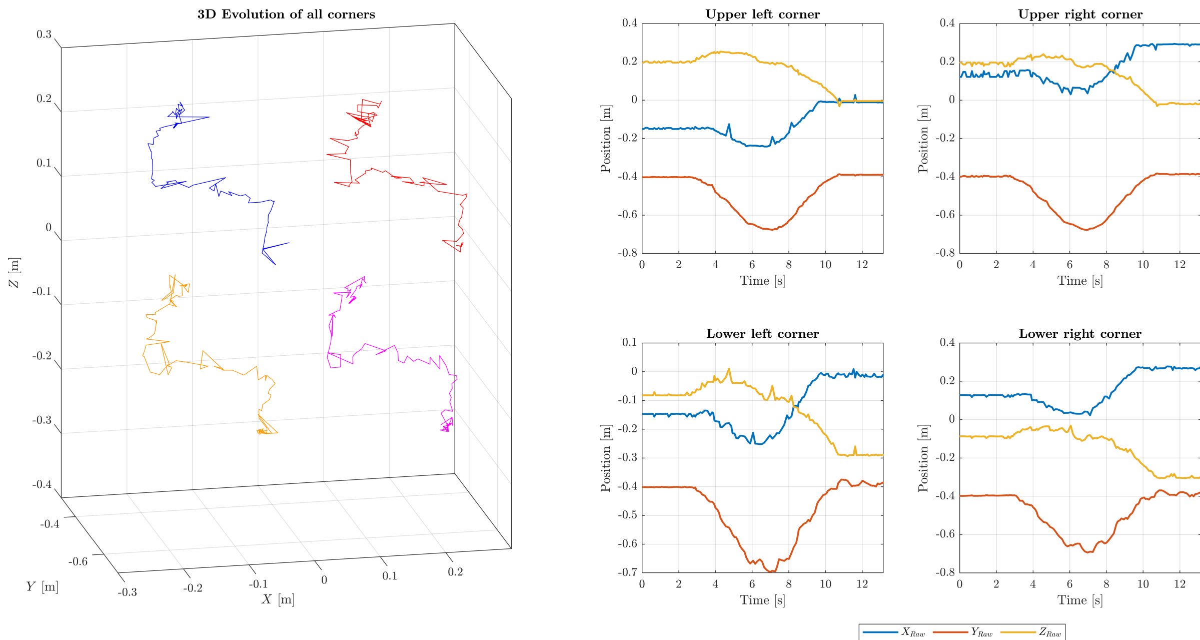

Figure 3.3: Raw data gathered through Computer Vision

First of all, we can clearly see how noisy the output is, which makes the obtained evolution not representative of that of the real cloth, and not fit to be our behavior to be learnt. Having the full evolution, we can discard outliers right away. These are all points far away from their two neighbors in time (previous and next points), which will be substituted by the mean between those. We can also apply filtering techniques depending on previous and future data points, which would cause a delay in a real-time implementation, but here we are just processing data to remove the noise on the full evolution. For example, we can apply a Gaussian filter with symmetric coefficients obtained with $w = \exp\left(-\Delta t^2/2\sigma^2\right)$ for a fixed σ and amount of time neighbors considered. The original point ($\Delta t = 0$) receives a weight of 1, and this value decreases for points before or after it. The new point is obtained with a weighted average inside the considered window, resulting in a much smoother trajectory, as shown in Fig. 3.4.

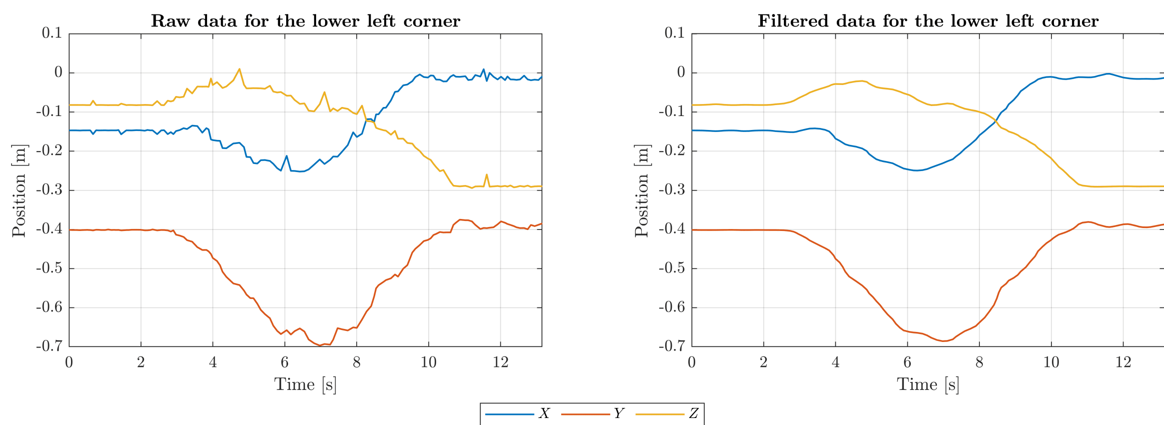

Figure 3.4: Comparison between raw and filtered data for the lower left corner

After this, the data must be regularized in time, as the output not only is slow in processing the vision data and giving the positions for all the nodes in the mesh, but it also returns data at irregular intervals. This can be changed easily with a simple interpolation in time, to obtain points at equally spaced intervals, as shown in Fig. 3.5, where the result has points every 20 ms. This is one of the most crucial steps, because the chosen time here will be the T_s at which the simulations must run during learning, and we must remember that the parameters to be learnt depend on this value.

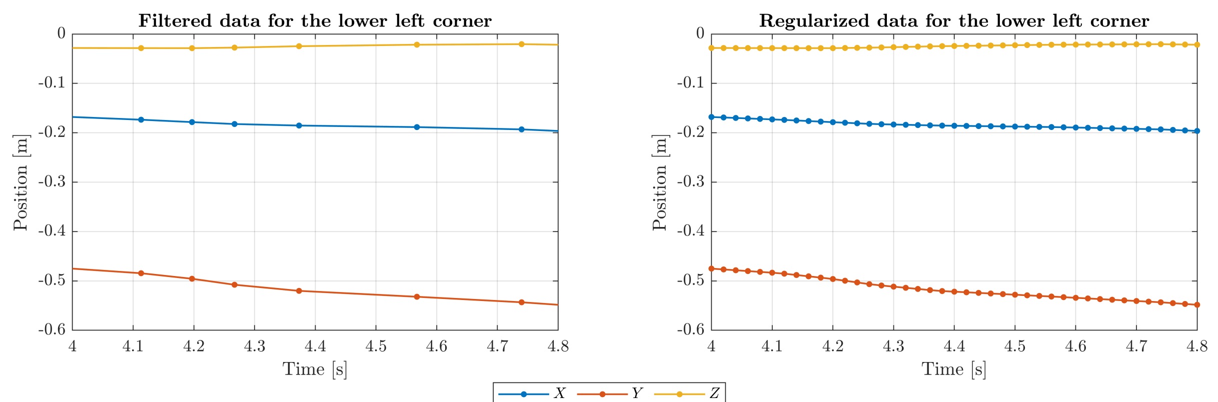

Figure 3.5: Zoomed in comparison between original data points and regularized ones

The mesh returned by the vision processing algorithm can have any arbitrary size, decided at the moment of execution. We can make this more flexible adding an option to interpolate in space to change mesh sizes. Even if this part of the code is functional and allows for several possibilities once the data is captured with a fixed size, it was only used after the first data gathering session, to test the codes with larger mesh sizes. After some learning experiments, more data was gathered, using meshes of (side) sizes ranging from 4 to 13 (16 to 169 nodes, to be extra clear), so this option was not needed anymore.

The last steps to process this data are trimming the periods before the start and after the end of the trajectory itself, leaving or adding some time with the cloth not moving in both ends, and computing the velocities with the simple Euler method (difference in position of two consecutive points over the time between them), with $v_0 = 0$, building a full state vector. The resulting matrix of state vectors over time is then saved in a separate file to be loaded during the learning experiments.

3.2.2 Learning with Real Cloth Data

Obtaining the optimal parameters for the linear model was done using REPS, following the structure seen in Algorithm 2. This means that a starting distribution is needed, with $\theta_k^0 \sim N(\mu_0, \Sigma_0)$ for all samples k of the initial epoch. We also need to execute a high number of experiments using the sampled parameters, and compute a reward from each one of them.

These experiments are open-loop simulations of the cloth moving subject to an input trajectory of the upper corners. This means that for each one of them, we only need to load the file containing the corresponding real cloth evolution (depending on session, trajectory and T_s), and also initialize and then simulate only the linear model. To make the evolutions as close as possible, the input trajectory of the linear model is set as the saved upper corner trajectory of the real cloth, and not the original TCP trajectory with some offset. The only restriction to mesh side sizes is that they must follow $n_{real} = p(n_{lin} - 1) + 1$ for some positive integer proportion p, to ensure we can use the mesh reduction function to select the correct nodes of the real mesh to compare to the linear one (e.g., we can use 10 to 4, 13 to 7 or 13 to 4. A graphical representation of this was shown in Fig. 2.20, the idea being the larger mesh contains all the nodes of the smaller one plus some more in between).

Once the experiment itself is complete, on one side we have the simulated evolution of the linear model, and on the other the saved evolution of the mesh representing the real cloth, reduced if necessary to the size of the linear one. The reward of the experiment must depend on how similar these two evolutions are, so computing the difference between their state vectors on each time instant is the first step to obtain it. Then, we use the square of the errors instead of their absolute value to penalize large differences even more, and average these values through time getting the Mean Squared Error (MSE) for each node. Taking only the difference in position, we could make the reward just the sum of negative MSEs, as all rewards must be negative and more negative is worse, but we can weigh these errors depending on their position inside the mesh, to penalize those on the lower edge of the cloth more than the upper edge, which contains the controlled corners, always with zero error. This yielded better results than penalizing only the lower corners and also than penalizing all the nodes equally.

This weighting can be done by multiplying the positions by a vector containing the corresponding weights. Considering the nodes are numbered from left to right, bottom to top, and that the meshes are squares, this vector must start with the maximum weight repeated n times, then decrease the weight for the next n, and so on, until all the nodes have been considered once, and the full sequence is repeated twice more to affect the Y and Z coordinates. Following these specifications, the weight for any position i can be obtained applying (3.5). This makes the errors on all three coordinates of the lower nodes have weight 1, and this value decreases evenly until the top edge of the mesh has weight $1/n$.

$$w_i = 1 - \frac{1}{n}\left\lfloor \frac{\text{mod}(i-1, n^2)}{n} \right\rfloor \quad \forall i \in \left[1, 3n^2\right] \tag{3.5}$$

If needed, the reward is then saturated at a minimum value fixed at the start of the learning process, as mentioned at the end of Section 3.1. With this and the parameters used on each sample, we can use REPS to find the updated means and covariance of our parameters.

The only thing left is the starting point, the initial policy or distribution to get the samples on the first epoch. For the case of $n = 4$, $T_s = 0.01$ s, the initial seed was set around the original parameters, at roughly approximate values. This was also tried as an initial policy for other sizes and times, but that resulted in an unstable linear model, and no successful experiments on the first epoch, which meant no policy update and an endless repetition of this epoch. Because of this sensitivity, an initial combination of parameters that resulted in a stable evolution had to be found heuristically reducing the variance, until the learning process could start from there.

Some final considerations can be made regarding the specific parameters being learnt in this process. The first one is that they can be divided in three types: stiffness, damping and length correction. The ranges of values for this one of this types of variables is fairly different, with stiffness reaching the hundreds, damping coefficients being around the units and the lengths being in the order of the hundredths (of their respective units). This is important when applying the factor λ, an exploration incentive, as a proportion of the mean of the singular values of the covariance matrix $Sigma$, because this matrix will have widely different variances for each type of parameter, resulting in a very large first singular value and a very small last one. Averaging them all together and adding a value proportional to that to the same covariance matrix can have no effect on the large values and be completely disproportionate for the small ones. To avoid this, all parameters are normalized to be in the same approximate range. In the end, the three stiffness values are divided by 100 and the length is multiplied by 100 in the distributions, so when a sample is taken, it must be multiplied by a weighting vector to obtain the real value.

The second consideration is that these values must always keep the same sign. By how the model is constructed, both stiffness and damping parameters are negative, and the length is positive. This potential issue is solved by simply discarding any sample that does not follow this rule, before executing its associated experiment, and getting new candidates until the condition is satisfied. Only then the simulation is called and the reward computed.

Now the learning process can finally begin. The main codes used can be found in the Appendices document. The final implementation allows to save all the relevant data after a given number of epochs, to pause the execution of the program without losing any value and without having to wait until the iterations are finished. The execution can be resumed again loading the previously saved files, and everything will be updated accordingly.

With this setup in place, several learning experiment sets were made. Each set corresponds to a considerably different situation (session the data is from, or real mesh size, for example) and contains multiple experiments inside, changing executed trajectory and sampling time. All experiments simulated 50 samples per epoch, but the total number of epochs changes from one to another, usually ranging from 100 to 200. The final total number of learning experiments executed with the objective of finding the optimal parameters of the linear model is 95. Removing all the experiments done with the nonlinear model instead of the real cloth, the ones done for testing and some repetitions, the number goes down to 84, which is still a very considerable amount. The task is now to study the results of these experiments, detect any outliers, check if they follow any pattern, and finally combine the valid ones to obtain a table of parameters depending only on size and sampling time, for the considered values.

3.2.3 Analysis of Results

The final amount of experiments is quite large, and all data had been saved, with a summary of the important conditions and results written automatically in a table. It would have been risky to keep executing simulations and completing learning experiments without checking results progressively, and without any kind of organization. The experiments were thus organized in sets, clearly divided depending on the used gathered data and the output mesh size. Knowing the linear model was going to be rather small to be able to simulate it in real time, in the end, all the learnt parameters are for models of sizes 4 and 7 (and varied T_s), to avoid aiming for larger models that were not going to be used, and because both can be obtained from larger meshes of size 13, which is very useful in the triple model scheme of the real implementation. To check results after every experiment, and even midway to decide to continue or stop manually, two kinds of plots were analyzed. An example of the first one can be seen in Fig. 3.6, showing the evolution of the means and deviations for all parameters.

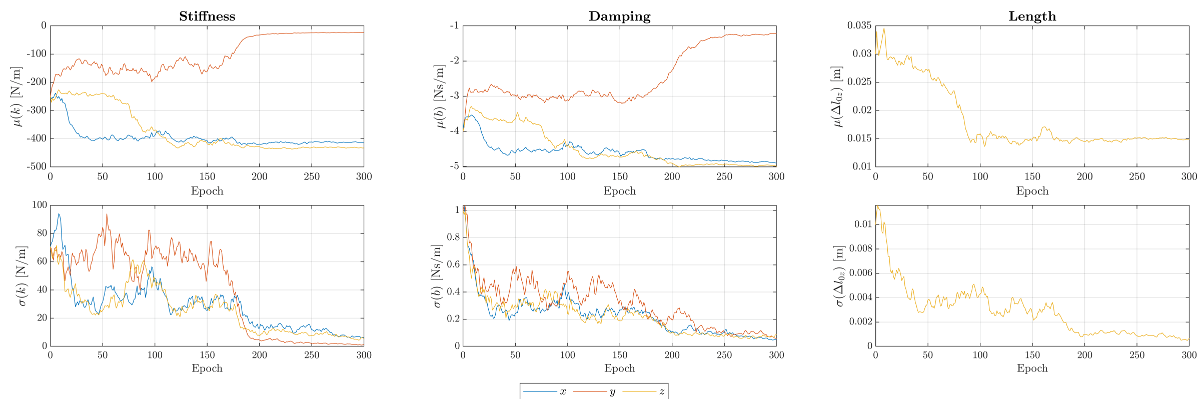

Figure 3.6: Evolution of means and deviations of all parameters in one learning experiment

ETSEIB

In the shown case (set 4, trajectory 11, T_s = 10 ms, n_{real} = 10, n_{lin} = 4), we can see how the means of the parameters change on the few first epochs and settle on some values rather quickly (Δl_{0z} being the exception, taking about 100 epochs). If only that was considered, the experiment could have been ended around epoch 150, after a quite steady evolution for all of them. Even considering the evolution of the rewards, shown in Fig. 3.7 for the same experiment, this could have been the case. However, we can see that at this point, variances (or their square roots, shown in the plot to keep the same units) are still considerably high for all parameters. This is why this experiment was continued, and not long after the 150 epoch mark, both k_y and b_y changed significantly, with a corresponding increase in the reward. With this change, the variances finally went down, and after some steady epochs, the experiment was concluded. In summary, some partial result analysis had to be made on every experiment, with the ending condition depending on both steady means and rewards, and low variances.

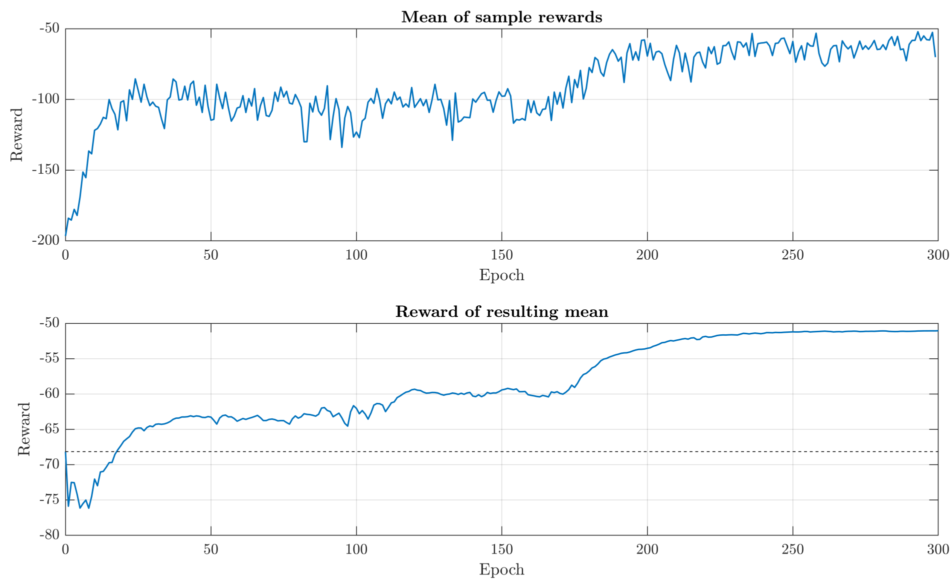

Figure 3.7: Evolution of the rewards for the same learning experiment

One important note about Fig. 3.7 is that there are two different reward evolutions depicted. The top one seems to oscillate a lot, and has an overall lower (worse) value, but both seem to follow the same trend. This is because, as the titles of the subplots clarify, the top one plots the result of averaging the rewards among all samples in each epoch, so even if some samples have very good rewards, having just one unstable simulation pulls the average reward considerably. Contrarily, the bottom one is the reward of executing the resulting updated policy, or more specifically the means of the parameters, without considering any variance. To give a clear example, we start with some μ_0, Σ_0, from which we take 50 samples $\theta_k^0 \sim N(\mu_0, \Sigma_0)$. We simulate for each one of these, and compute the associated reward r_k^0. The mean of these 50 rewards is what is shown in the top plot. With the previous distribution and these rewards, the policy is updated using REPS, obtaining μ_1, Σ_1. We can now make an additional simulation with $\theta_1 = \mu_1$, and the resulting reward will be shown in the bottom plot. In fact, we also have an initial reward simulating with $\theta_0 = \mu_0$, so the bottom plot always has one point more than the top one (like in the fence-post problem, where there is always one post more than the number of fences in between).

This distinction is important because, even if ideally at the end of all learning experiments we would reach zero variance, this is never exactly the case, but given we need some exact, constant parameters, the final values taken will be the final means, and not the entire policy or distribution.

With this considerations in mind, we can now proceed to analyze the final results of all the learning experiments. In the end, the considered sampling times were T_s = 10, 15, 20 and 25 ms. Less than the original 10 ms has no purpose, as it is already fast and we need more room to complete the optimization within a step. Experiments with 30 ms were also tried, but no successful simulation was completed with any of the several initial policies tested, making this sampling time too large to simulate. Besides that, we have already mentioned how the only two considered linear model sizes are 4 and 7, so all the experiments use one of the 8 possible combinations of these parameters.

As described in Subsection 3.2.1, a total of 4 different trajectories were executed with the real robot to capture the evolutions of the real cloth. The idea was that different trajectories could capture distinct behaviors of the cloth, as they were quite different in shape and/or speed. The dependency of the parameters on mesh size and sample time is already known, so to isolate the effect of the trajectories, we must compare between them on the same combination of n and T_s. As an example, Fig. 3.8 shows this comparison for $n = 4$, $T_s = 10$ ms, with a subplot for each type of parameter ($k,\ b,\ \Delta l_0$). Every dot is the final learnt value of a parameter, with all experiments done with the same trajectory aligned vertically, and they are colored according to the final reward obtained in that experiment simulating the final means. For an easier identification, the absolute value of the rewards (which are always negative) is used, so we can talk about "costs" instead, as higher is now worse, and the best values are still the ones closer to 0 (blue). These costs have units of cm^2, but we must remember they are the MSE of all nodes multiplied by some weights and added together, so a cost of 100 does not mean an average error of 10 cm, but rather about 1 cm in a 4×4 mesh, considering the weights used and that two nodes always have null error. This is why the units will be left out for rewards and costs in general, as their direct meaning is lost.

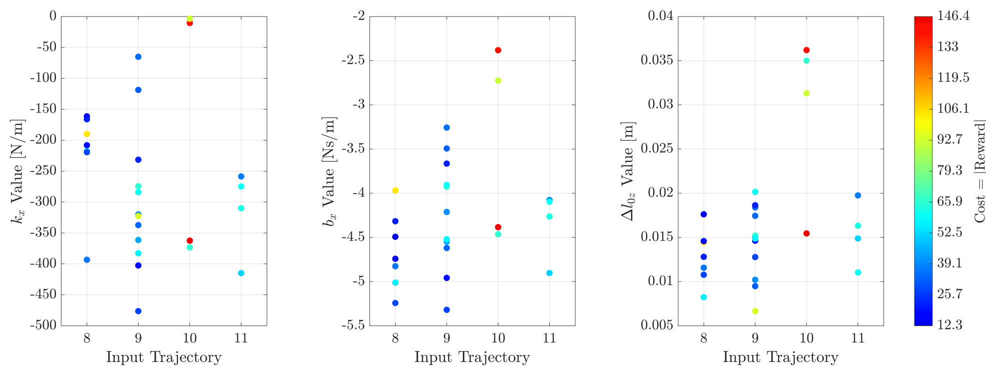

Figure 3.8: Comparison between learnt parameters and their rewards depending on input trajectory

In these plots, and the equivalents done for other times and sizes, we can notice a common trend. The obtained values and rewards for trajectories 8 and 9 are similar, given these trajectories share the initial section, only differing in speed after a while, and in the ending movement. Trajectories 10 and 11 give values a bit further away in some instances, but while the rewards of the experiments done with

trajectory 11 are similar to those of 8 and 9, the rewards obtained with trajectory 10 are consistently the worst ones (higher costs, red dots). In fact, this was somewhat expected after seeing the behavior of the robot performing this trajectory (the fastest one) and the output from the vision algorithm, which struggled to keep up, resulting in the most unreliable evolutions compared to the rest of the gathered data. This is also why there are fewer experiments done with this trajectory, as some of them failed to find a combination of parameters that resulted in a stable simulation, and some others were not even started following the exposed reasoning and the obtained partial results at the time.

Removing all the results obtained with trajectory 10, we are still left with 74 experiments. Seeing how for the remaining trajectories the obtained parameter values are within the same range, and the rewards are much more similar, the differences produced by the input trajectory are deemed negligible, as they were theorized to be. With this distinction gone, we can now focus on the two variables that affect the parameters, mesh size and sampling time. As a side note, to explain the role that the mentioned experiment sets play in all this, they are, at the core, divided by mesh size, each one corresponding to a different combination of original captured mesh (13, 10, 7 or 4) and final linear model size (7 or 4), so this classification fits nicely with one of the two variables considered.

With only these two dependencies left, we can plot the obtained results against them first, to compare each possible combination separately and also by groups. This is all done in Fig. 3.9, where on the left plot we have each one of the 8 combinations separately, and then grouped by size or time step on the right. In this case, the vertical axis represents reward, and the color is also related to it, but not tied to an overall scale, instead simply going from blue to red for each combination, to clearly see the best and worst cases and their correspondence between the three shown plots.

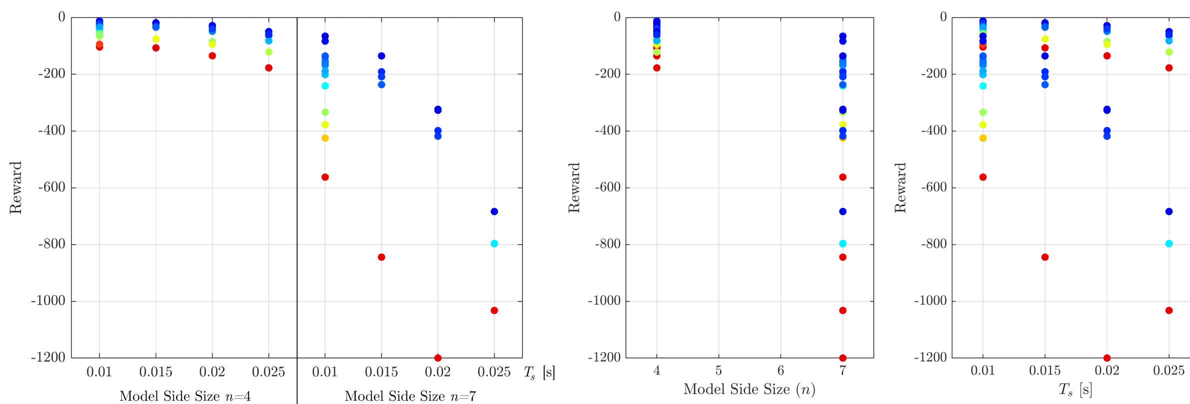

Figure 3.9: Rewards for all the considered experiments, grouped by n and T_s

The most obvious difference is found between the two mesh sizes. For all the considered time steps, the experiments with $n = 4$ have significantly better final rewards, with not only better values in the best cases, but with a smaller range between the best and the worst results in all cases, even with slightly more experiments completed for this size (43 versus 31). This is also seen in the right-most plot, where the red dots near the top correspond to the worst results with $n = 4$, and we must go down almost 500 points to find the best result using $n = 7$ in the worst case ($T_s = 25$ ms).

A decrease in reward going to a bigger size was to be expected, as the errors are added for all nodes and not averaged, giving a worse reward in absolute value on larger meshes by default, but, first, these are the final rewards after the whole learning process, and second, some of them are near the rewards for some $n = 4$ experiments, so the most surprising result is the range these rewards take.

Besides this difference, increasing the sampling time has almost no effect for the smaller size, with the rewards having just a slight downwards tendency, but this is greatly amplified for $n = 7$, where the rewards clearly go down as T_s increases. We will still consider these combinations, as the best results still have reasonable associated rewards (some blue points actually have multiple experiments behind, with very similar rewards), but we must not forget this plot, and keep in mind that the parameters for the larger size and longer time steps might not be the most reliable either. A possible justification for this is that we are using a linear model, which was created to work under very specific conditions (T_s, n and trajectory), with the modifications to work with multiple sizes being made in this Thesis (Subsection 2.5.1) and not originally. Even if this is not a typical model linearization around a working point, as we move away from the original conditions, it is not strange to find that the model does not adapt as well to them.

To get a unique set of parameters for each combination, we could just average the results or use a median, that is more robust to extreme values, but being able to access the rewards for each experiment, we can use this information again to compute a weighted average, giving more weight to the best results. A representative example of this situation is shown in Fig. 3.10, for $n = 4$, $T_s = 10$ ms. Here the vertical axes are the values of the parameters themselves, separated in 3 plots according to their nature, as their ranges are quite different. They are colored with the same scale according to the final cost of the experiment they come from, and thus the color is proportional to the weight they have in the computation of the values of the final parameters.

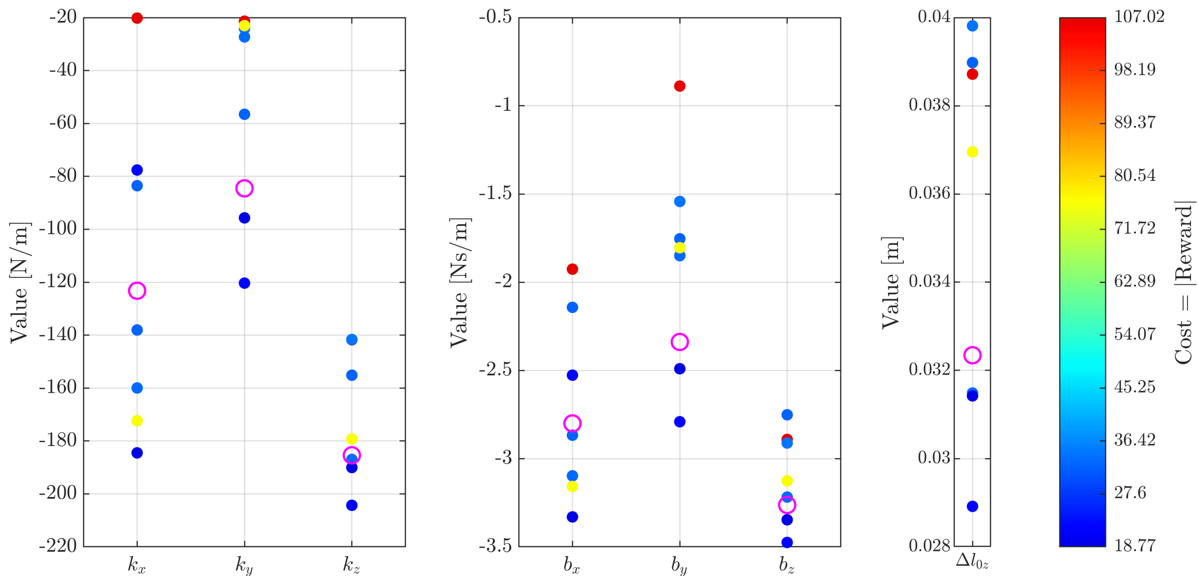

Figure 3.10: Example of computing the final model parameters for $n = 4$, $T_s = 10$ ms

In this case, we can see how there are many results in blue, indicating good rewards (around 5 mm of average error, if we consider size and weights), and just one or two outliers, with parameter values quite far away and the worst associated rewards. The final weighted average, which is actually done using relative weights d_k computed exactly like the ones in the REPS algorithm, gives a combination of final parameters (marked with a magenta circle) near the concentrations of blue points.

The final learnt parameters for all the considered cases are shown in Table 3.1. We can see how, for the original case of $n = 4$, $T_s = 10$ ms, stiffness and damping in the Y-axis have increased significantly in absolute value (they were -13.4221 and -2.5735, respectively), but they are still the lowest of their respective kind. It is also interesting to see how the stiffness and damping parameters are closer to zero both with the larger mesh size and with longer time steps, which makes sense physically too, as with more nodes the springs are shorter and do not need to be so rigid, and with more time the displacements allowed will be larger. Additionally, these parameters are multiplied by T_s in the construction of the model, so they seem to try to compensate. The length correction parameter goes in the opposite sense, being larger for bigger sizes and time steps. This can be related to the stiffness in Z being lower, as this change will increase the super-elastic effect of the cloth being stretched under its own weight, and thus the correction factor must be more prominent. However, this length increases to the point of being larger than the cloth side length for $n = 7$, $T_s = 25$, which contributes to the point made about the results for this combination being the least reliable ones, together with the fact that the stiffness in X is even lower in absolute value than the one of the Y-axis for these situations. Even then, simulations can be run with these parameters, for multiple and varied trajectories moving in all directions, and they do finish with a stable evolution and reasonably good tracking.

Table 3.1: Final linear model parameters, depending on size and T_s

Size	T_s **[ms]**	k_x	k_y	k_z	b_x	b_y	b_z	Δl_{0z}
4	10	-325.8100	-166.2000	-395.4000	-4.4870	-3.4089	-4.8105	0.01493
4	15	-123.2300	-84.5230	-185.4100	-2.8007	-2.3387	-3.2631	0.03234
4	20	-41.8000	-39.1880	-91.4970	-1.7286	-1.3309	-2.2554	0.07095
4	25	-25.2390	-28.0140	-56.5340	-1.2255	-1.2131	-1.6997	0.11614
7	10	-158.6600	-101.5600	-295.8400	-3.2529	-2.9498	-3.9572	0.09723
7	15	-49.6420	-110.2600	-186.1100	-1.8422	-2.2991	-3.0405	0.14927
7	20	-33.7230	-61.2510	-112.6700	-1.3052	-1.5734	-2.3744	0.24107
7	25	-35.4460	-38.5820	-71.7900	-1.3422	-1.1800	-1.9108	0.38143

As a final comment, with all this data obtained from the learning experiments, we could think of building a model that fits it (using regression, for example), returning a combination of parameters given some n and T_s between the tested values or even extrapolating further from them. However, given the proven complexity of this system, the nuances of such a regression model, and that we have obtained a sufficient variety of possibilities for the experiments needed to be done for this project, this is left out of the scope of the Thesis, and as possible future research work.

3.3 Learning the Parameters of the Controller

This section has been made possible thanks to the changes introduced to the controller in Chapter 2 and the new parameters of the linear model found in Section 3.2 for different conditions. The new controller has now a large number of parameters to be tuned, if we count weights, bounds and all the different categorical possibilities described in Subsection 2.5.7, which, even if some of them make more sense theoretically in one way for a better tracking, we are trying to strike a balance between minimal error and computational time, which act against each other, so we can apply RL to the controller and obtain results for all these options.

One example of the new possibilities we can simulate (and later execute on the real robot) is shown in Fig. 3.11. This simulation was executed with T_s = 20 ms and a COM side size of n = 7, which is one of the most demanding situations for the model, and we can see how even then, the tracking is impressively good. The KPIs demonstrate so, with a tracking RMSE (“Norm” method) of 11.3 mm and an average computational time per step of 28.9 ms.

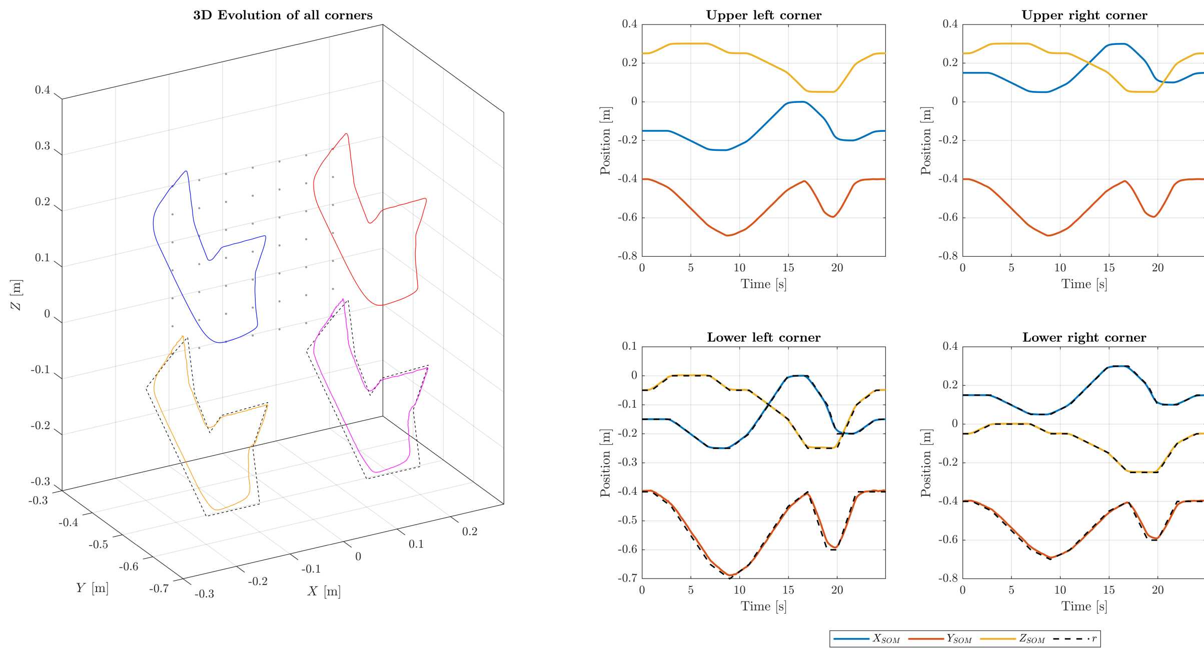

Figure 3.11: Simulation results with a longer trajectory in more demanding conditions

The increase in error with respect to the previously executed simulations can be justified by the new trajectory and conditions, and we can see how the differences between reference and lower corner trajectories are localized in the hard, fast turns when returning back to the initial position, a result that matches the expected behavior when minimizing slew rates. However, one might think the increase in computational time is unreasonable, given how times lower than 10 ms were reached before. This is actually not the case, and brings us back to the topic of tuning and the amount of parameters that can change. These almost 30 ms were obtained using H_p = 25, and for a T_s of 20 ms correspond to about

1.5 times the step, which is equivalent to the 15 ms obtained with $T_s = 10$ ms and $H_p = 30$ seen back in Table 2.1. We can see how the proportion is similar with also similar prediction horizons, with the difference being the total prediction time, increased from 0.3 to 0.5 s with the new conditions. With just reducing the horizon to $H_p = 20$, the average time goes down to about 14.4 ms, already under the time step of this simulation, with almost no increase in error (less than 1 mm) and with a total prediction time still over the original one (and much more than the 0.15 s needed to get times under T_s when this value is 10 ms, as analyzed in Subsection 2.5.5). And, of course, these are not the only values that affect the results. For example, both weighting matrices of the objective function, Q and R, had to be tuned again. Done manually, this process takes long and is based on trial and error until finding the combination that yields the best results, so it is the perfect opportunity to apply RL.

This section is divided in three parts. The first one (Subsection 3.3.1), describes all the parameters on a closed-loop simulation that could be potentially learnt. The second part, Subsection 3.3.2, contains an analysis of the categorical parameters deciding the exact structure of the controller, and finally, the best option is then tested to obtain the optimal overall results in Subsection 3.3.3.

3.3.1 Initial Considerations

Before starting the learning experiments, we can take a look at all the parameters that affect the outcome of the closed-loop simulations, identify them and their nature, and decide the best course of action. First, we have the mentioned sampling time T_s and prediction horizon H_p. They can only take discrete values, as the linear model depends on the former and we only have learnt parameters for a select few values, and the latter can only be an integer (number of steps). For this reason, and also because their final values will depend on the conditions of the real implementation, we will not use them as parameters to be learnt, but instead we will try to learn the best overall combination of the rest of parameters applicable to all possible situations. In fact, a dedicated analysis has been made comparing real results obtained for multiple combinations of these parameters, shown in Section 5.3.

Next, we have the objective function weighting matrices, Q and R. Until now, for Q, we have used the adaptive weight Q_a, depending on the direction of the reference inside the horizon, multiplied by a constant value Q_k, but we could try disabling Q_a and check its effects. Furthermore, both matrices multiply vectors of 6 components (3 coordinates of the 2 corresponding corners), so they have size 6×6, but until now we were using the same weight for all components. Weighting the same direction on each corner differently makes no sense in the considered application, which wants to track both trajectories and moves the cloth with one robot, but we could still consider 3 different weights on each matrix, one per Cartesian space coordinate. Theoretically, this approach could make the optimal weights depend on the executed trajectory, so it might go against finding the best overall case. However, it could be a faster substitute for the adaptive weights, and we could apply different pre-computed weights depending on the situation checking longer sections of the future reference. Of course, this would be better only if the computations are actually faster, there is a clear relationship between trajectory and optimal weights to select from few optimal cases, and these cases offer significantly better results if applied in their specific scenario rather than using a more general combination of weights.

An important remark about these weighting matrices is that what actually matters are the relative weights, the proportions between these values. Having $Q = 2,\ R = 1$ makes all components inside each matrix matter the same, with the errors having double the cost of the control signals (or slew rates), and the same can be said about $Q = 20,\ R = 10$. With them being in a function to be minimized, all constant factors do not affect the final optimal solution, only the value of the objective or cost function at that point. If we multiply all weights by a constant, the optimal value of the objective function will be multiplied by that constant, but the optimal solution itself will not change. This is important now, because if we consider all the weights as independent parameters to be learnt, we would be considering an additional Degree of Freedom (DoF) that is not actually there, and we might get optimal results for a wide variety of combinations (high variances), sliding over the free parameter. Therefore, we need to apply some restrictions to these values.

When they have the same weight for all components (each matrix is a constant times the identity), we can represent them in a 2D plot to have a clear visual representation of their behavior. Fig. 3.12 shows multiple ways to represent the two values where once one is fixed, the other will be too, so the free variable is their relative value. This proportion is constant along any line crossing the origin, like the shown red one, so one initial way of having a unique free parameter would be to use the angle θ with the horizontal axis (in this case Q), ranging from 0 to $\pi/2$ rad, with $Q = \cos\theta$, $R = \sin\theta$. A very similar but simpler alternative is the one depicted in (a), where $Q + R = 1$ and they are always in the shown diagonal line. While this is linear and we can easily see which parameter is larger, the exact proportion is not directly clear. One way to solve this is fixing one of the two to always be 1, like in (b), where $Q = 1$ and R is free, either smaller or larger. The only disadvantage of this is that the non-fixed parameter is not bounded, and this can become a problem if the optimal proportion is very large (orders of magnitude apart), as depending on the starting Normal distribution, we might not reach it in a reasonable amount of iterations. This is solved by forcing to 1 the maximum of the two values on every case, instead of always the same one, like shown in (c). This way, the parameters are always sliding in two edges of a finite square from 0 to 1, where their proportion is still clear to us without needing to operate, and with even extreme proportions being easily reachable for the learning algorithm, setting the lowest weight around 0. The final considered option, (d), forces the minimum of the two to 1 instead, but this results in two infinite edges, with the same problems as (b). Following this reasoning, the weights of the objective function will always be set according to (c), with the largest one always being 1.

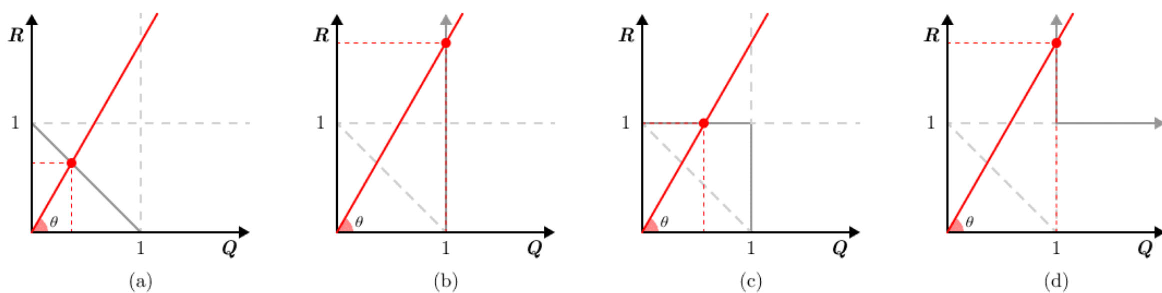

Figure 3.12: Different possibilities to represent two weights with the same proportion

More specifically, this will be true for all the generic cases: Q and R being two scalars times the identity, considering different weights per coordinate but with $Q = R$ (3 weights in total, but one will be forced to 1 and 2 will be really free, between 0 and 1, in any given situation), with one matrix being a scalar times the identity and the other having 3 different weights (4 in total, 3 free), or with all 6 weights being different (5 free). With 3 total parameters, we can think of them as being in the three faces of a unitary cube defined by a weight being 1 and the rest equal or lower to 1. A final case we can consider is having different weights per coordinate and $Q \propto R$ instead, with a linear proportion to be learnt too.

Moving on, after the weighting matrices we have another categorical binary variable representing whether we consider minimizing the control inputs u or the slew rates Δu. The benefits of using the slew rates have been discussed, but with the new vast array of options, there might be some cases where not using them benefits greatly, for example with lower computational times.

Finally, we have the bounds considered in the optimization problem. All the optimization variables are control signals representing displacements in a time step. The bounds are only there to ensure the linear model does not extend past a reasonable value unreachable by the nonlinear model or the real cloth. The exact limit or expected maximum value depends on the maximum slope of the reference trajectory, as the cloth must move approximately at the same rate (considering the reference is on the lower corners and the control signals on the upper ones), and there is no need for much more. Not only that, but if the simulation unstabilizes, even with bounds as low as the required slope, the evolution can be chaotic nonetheless, and if the simulation is stable, these bounds only increase the computational time if not quite relaxed. The other bounds in the problem come from the constraints, concretely the constant distance between the upper corners imposed by using only one robot and introduced in Subsection 2.5.3. These are equality constraints, so the bounds are set to 0. Relaxing this to obtain faster computations comes at the price of precision, and considering small relative movements between the two controlled corners, which will affect the resulting TCP pose. In the end, both bounds will be fixed and not considered as parameters to be learnt for the aforementioned reasons.

To summarize everything explained in this subsection in a short list of parameters, we can consider:

- The time step T_s and prediction horizon H_p, but they will be analyzed with experimental results due to their discrete nature and how results depend on the real conditions.
- Using the adaptive Q_a or not to obtain Q.
- Minimizing either the control signals u or the slew rates Δu.
- The structure of matrices Q (or Q_k) and R: a scalar times the identity, a different weight per coordinate and equal or proportional to one another, or another combination.
- The weights of these matrices themselves.
- The variable bounds, but they only increase either error or time and have been discarded.

3.3.2 Obtaining the Most Suitable Structure

From the list at the end of the previous subsection, there are several categorical and binary variables, that rather than being parameters to be learnt, must be analyzed comparing learning experiments using all possible combinations. To that end, the several possibilities for structures of the weighting matrices were reduced to just 3: two scalars (multiplied by the identity), Q with different weights per coordinate but with R being the same for all of them (4 weights in total), and having different weights for each coordinate for both matrices but with $Q \propto R$ (also 4 weights in total).

A first round of experiments were conducted using $n = 4$, $T_s = 20$ ms, $H_p = 25$ and following the same reference trajectory. At this point in the Thesis, the real implementation had progressed enough to test some closed-loop trajectories too, and these conditions had one of the best observed behaviors. Additionally, to reduce the time needed to execute these much slower closed-loop simulations, the simulations will be done with a linear SOM. This means that the conditions are also closer to those of the real implementation, where there is a linear "backup" SOM. The triple model scheme in simulation, that would be the closest to reality, is sadly the slowest option, as the evolutions of the nonlinear model, even if discounted from the computational time because they will not be present in reality, are the ones that take the longest to compute. However, a few simulations have been executed to check that the tendencies are the same and the results of the performed analysis apply all the same.

Given that we might get times over T_s, two reward functions were tested. One with just the negative RMSE ("Norm" method) with respect to the reference trajectory, as an equivalent for our usual first KPI, and another also penalizing the time spent over the maximum allowed, but not times under T_s, called TOV. The reward was not changed to this second option without testing, in case adding a secondary objective with no additional relative weight was detrimental to learning the parameters that yield the minimum tracking errors, which is the primary objective.

All these options give us a total of 24 experiments to complete in this first round: two options for Q_a times two for Δu times three for the structures of the weighting matrices times these newly introduced two reward function variants. The parameters to be learnt iteratively using REPS are the weights inside the matrices themselves, either 2 or 4 values depending on the structure.

The purpose of this first round is to analyze the differences mainly between the two first binary categories, and check if the best option between them is consistent among all types of weights and reward functions, while trying to notice any kind of trend or pattern in the resulting optimal weights for each case. To analyze the differences between weight structures, we need to execute multiple different trajectories, as mentioned before, so this requires further experiments in the following rounds. However, this first one can also help us to decide the best reward function to use between the two candidates.

As mentioned previously, the learning algorithm used is REPS, explained at the end of Section 3.1, with codes adapted for the new situation (closed-loop simulation files converted to functions depending on the input parameters and returning a reward), but the situation is completely analogous to that of Section 3.2 where we learnt the parameters of the linear cloth model.

We can first divide the results of this first round depending on the KPI we are analyzing, RMSE or computational time. Then, we can organize them in groups of 4 with the same weight structure and reward function, and make comparisons purely based on the use of Δu and Q_a. This is shown in Fig. 3.13 and Fig. 3.14, with results colored according to tracking RMSE in the former and computational time (relative to T_s) in the latter.

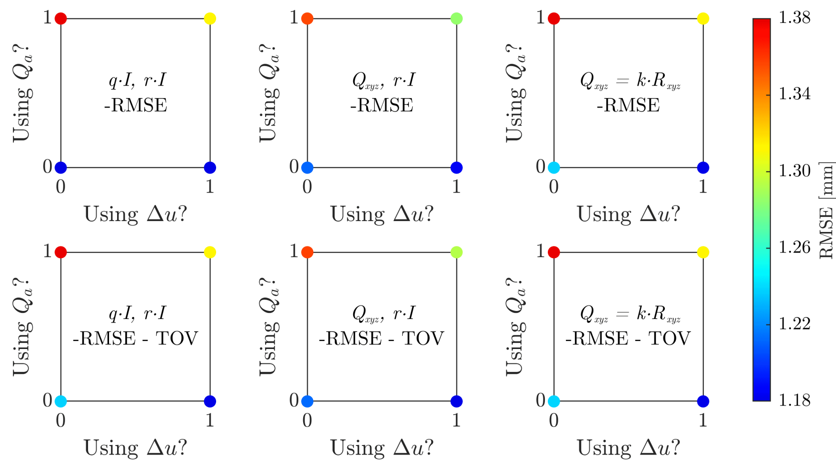

Figure 3.13: Results of the first round of learning experiments (RMSE)

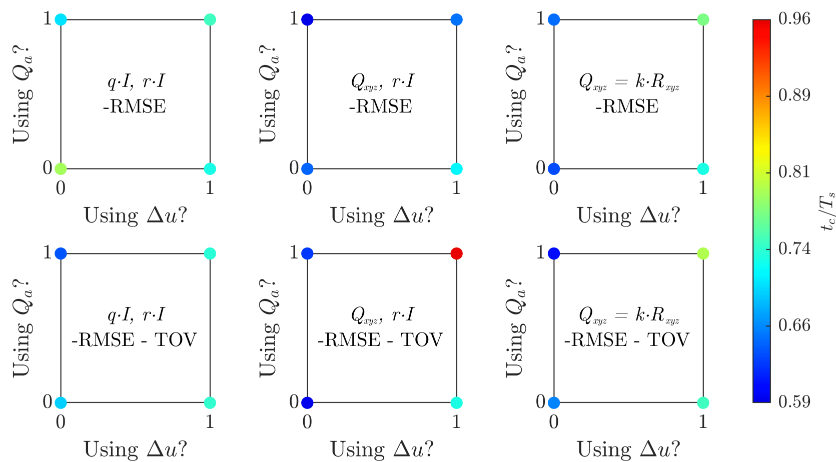

Figure 3.14: Results of the first round of learning experiments (time)

First of all, looking at the tracking errors, we can see how regardless of weight structure and reward function, the worst results are always found using Q_a and u, and the opposite selection, with only Q_k but with Δu, yields the best outcome. Looking at the time plots, the most common effect is that computational times increase slightly with the use of Δu. However, as we can see in the marked values of the color scale on the right, absolutely all optimal results have a computational time lower than T_s (20 ms in this case), making the negative impact negligible, and the best choice clear: the best results are obtained without using Q_a but minimizing Δu.

In fact, having all results with computational times under T_s means the reward functions were the same in most cases, with the term penalizing times over T_s not being active. Indeed, comparing the resulting learnt weights obtained with the two functions for each of the three structures, we can see how they are really similar, as shown in Table 3.2. The obtained rewards, as well as the KPIs separately, are also almost the same (about 0.3% of difference). This is not to say that the new reward function has absolutely no effect at any point, as the starting epochs have samples with combinations of parameters that increase the computational time to levels above T_s, and the additional term helps avoiding these values, potentially reaching optimal results sooner. However, the time measurements are sensitive to memory and CPU usage, and for such a minimal improvement, considering how long the learning experiments take anyway, the reward function penalizing time is discarded for future experiments. Additionally, if we used it for a general case, we would have to study the relative influence of each term, and weigh the two objectives they represent, better tracking or faster optimizations, to find the best balance. For now, we can focus in obtaining the best possible tracking, as that is the primary objective of the controller, after all.

Table 3.2: Optimal Q, R weights, using Δu and no Q_a, for all other combinations

Reward	**Weights**	Q_x	Q_y	Q_z	R_x	R_y	R_z
−RMSE	$qI,\ rI$	1	1	1	0.1707	0.1707	0.1707
−RMSE −TOV	$qI,\ rI$	1	1	1	0.1695	0.1695	0.1695
−RMSE	$Q_{xyz},\ rI$	0.9159	0.7587	1	0.1885	0.1885	0.1885
−RMSE −TOV	$Q_{xyz},\ rI$	1	0.7215	0.9576	0.1676	0.1676	0.1676
−RMSE	$Q_{xyz} = kR_{xyz}$	1	0.8892	0.4362	0.1791	0.1593	0.0781
−RMSE −TOV	$Q_{xyz} = kR_{xyz}$	1	0.8057	0.6741	0.1823	0.1469	0.1229

Another interesting result obtained with these experiments is that the final tracking error is approximately the same regardless of the chosen weight structure (less than 0.5% difference between best and worst), and the minimum errors are actually obtained with the first and simplest option. This means that the more complex structures, with weights depending on direction, actually do not adapt better to the trajectory leading to smaller errors. In fact, looking at the results of the first round, the consistency of obtaining worse results using an adaptive Q_a is also an indicator that adapting these weights to the current situation might not be the best decision overall. Of course, only one trajectory has been tested, but this reasoning, the slightly better results, and especially simplicity (not having to switch between optimal weights for specific situations, if optimal solutions really depended on the trajectory) push us to select the first structure, with just two weights.

Now we must check that the chosen options, which have been analyzed with results obtained using a linear SOM, are also the best for the real triple model scheme, and that all the decisions made are the best course of action for the real implementation too. To that end, and keeping in mind that the 24 experiments were done with a linear SOM to reduce the overall execution time, we will not repeat all 24 again for the different scheme, but just 4, for the first weight structure and the original reward function, to check the effects of Q_a and Δu. The results for both KPIs are shown in Fig. 3.15.

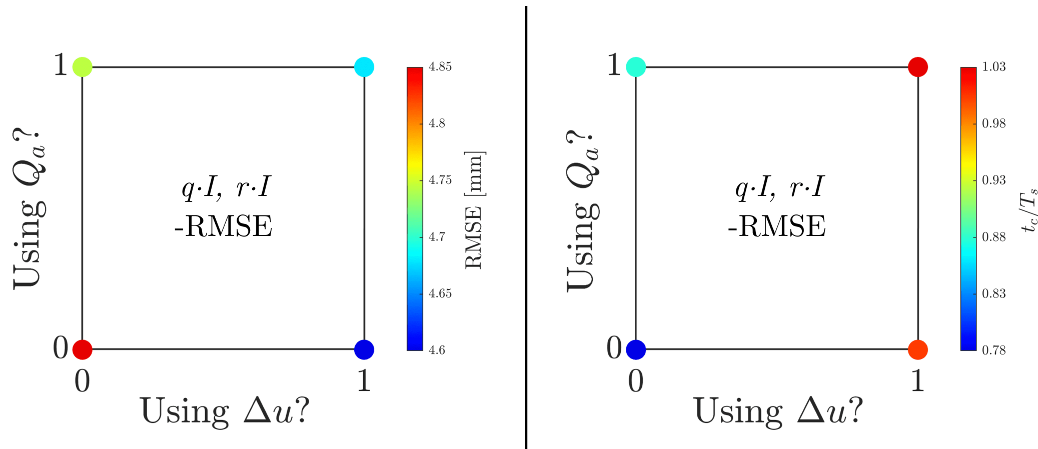

Figure 3.15: Results of the learning experiments using the triple model scheme

It is clear how the results match with the ones obtained previously. The best combination to minimize the tracking error is still using Δu with no Q_a, even if these errors are larger overall due to the evolution of the nonlinear model representing the real cloth not matching the one predicted by the linear COM as well as before. Regarding computational times, the most notable impact is still the increase when using Δu, with the worst case going slightly over the maximum allotted time of 20 ms. Thankfully, not using Q_a reduces this time a bit, but still being on the very edge of T_s. This could probably be improved using the reward function with the TOV term, and in any case we must remember the times are approximate and dependant on the tasks running in parallel with the learning experiments, so the option with the best tracking is still the chosen one, and the correspondence with the previous simulations is clear.

At this point, we can confidently say which is the best structure for the controller: it does not use Q_a, the objective function penalizes Δu, and each one of the weighting matrices Q and R has the same values for all coordinates, so there are only two weight values. This consistently gives the best tracking results without increasing computational times over the limit, and is the simplest alternative without sacrificing performance. However, the work is not completely done, as we have not analyzed the final optimal weights themselves and the specific tuning of the controller, and how it is affected by different trajectories and conditions. This is done in the following subsection.

3.3.3 Tuning the Resulting Controller

After the process described in the previous subsection, we have settled between all the possible options for the controller, but still have not analyzed how different conditions, like multiple trajectories, H_p, or T_s, affect the tuning itself. This is different than the analysis done with experimental data on Section 5.3, where the changes in the latter two variables are compared against the tracking KPI. Here, the intention is to find if the optimal tuning is the same regardless of these conditions, or if they affect the optimal Q and R weights. Luckily, the final structure of the weighting matrices is a scalar times the identity for each one of them, so we only have two values to learn. In fact, what is important, as discussed in Subsection 3.3.1, is only the proportion between these two values, so there is just one value to be learnt.

With this last consideration, we can adapt our approach to learn this value in different conditions. We know that, for the studied case of the previous trajectory and $T_s = 20$ ms, $H_p = 25$, the optimal weights are approximately $Q = 1,\ R = 0.17$, or a ratio $R/Q = 0.17$. However, we do not know how the error depends on this parameter in its full range. The learnt value could be a local optimum, it could be in the middle of a wide region of similar results, or next to a sudden increase in error if it varies slightly.

To get a better idea of the distribution of resulting errors versus R/Q value, we have two main options. The first one is to sweep the entire range of possible values at a fixed step between two consecutive points tested, and check the results. The other option is to apply RL using REPS as before, but just for a couple of epochs and with a lot of samples for each one, to obtain results for the whole range of possibilities. We can first check how both methods are equivalent with a single test for each one. The advantage of the sweeping method is that we choose the exact points to simulate, and we can create a uniform distribution to start visualizing the results. Furthermore, while at it, we can also simulate points with the same R/Q ratio but without one of the two being 1, as we forced until now, to empirically demonstrate that what was shown in Fig. 3.12 actually applies in practice.

The plots shown in Fig. 3.16 are the results of simulating the same trajectory as before, with the same conditions ($T_s = 20$ ms, $H_p = 25$, $n = 4$), but for all weights Q, R in steps of 0.05 along the two outer edges of the unit square. The dots are colored according to the obtained errors (saturated at 10 mm), as indicated in the color bars. On the left, all the results are shown, and we can clearly see a sudden jump from red to blue around $R = 0.2$. Quantitatively, it goes from 19.83 to 1.19 mm of RMSE within two steps. We can see another red dot when $Q = 0$, as we could expect when not penalizing errors at all. Removing the worst results we obtain the central plot, where the small differences can be appreciated. The error progressively goes down as R/Q decreases, until R is too low and the aforementioned jump happens. On the right, we added some points with the same R/Q proportion, to show how all of them result in the same tracking errors.

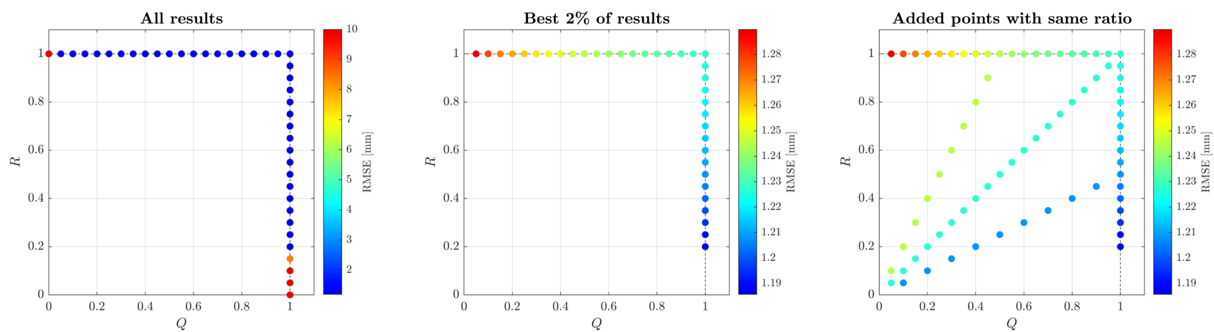

Figure 3.16: Results obtained by simulating across all possible Q, R values

The most important result here is the central plot. We have a clear progression of the tracking error, reaching its minimum value just before the system unstabilizes for not penalizing the slew rates enough. The error decreasing as Q increases with respect to R is what we would expect theoretically, as it is the weight that penalizes the tracking errors themselves on the objective function inside the MPC.

We can now compare these results to the ones obtained with two REPS epochs simulating 100 samples each. Here we can plot the results of both epochs to see the evolution of the distributions, and the best results to appreciate the differences between the best ones. This is precisely what is shown in the three plots of Fig. 3.17, where the dots are colored according to errors saturated at 10 mm, as before.

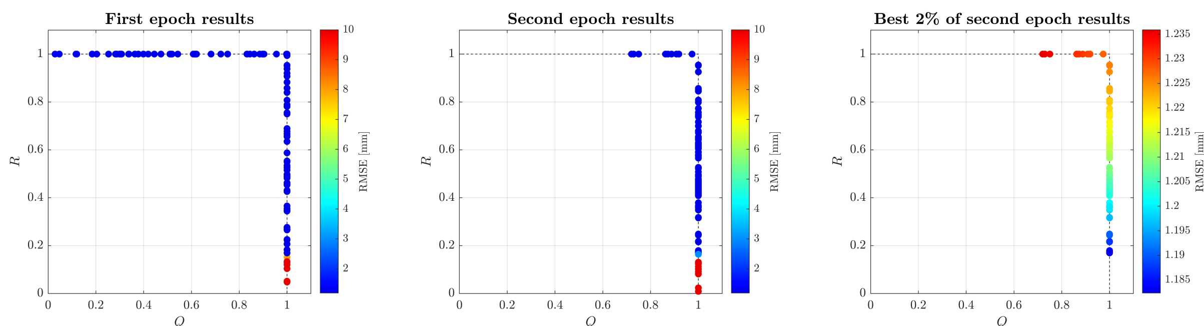

Figure 3.17: Results obtained by executing 2 epochs of REPS with 100 samples each

Clearly, the results are the same as before, with the added benefit of the second epoch moving the distribution towards the best results and removing all samples for low Q values, making the error range narrower (the red dots here have approximately the same error as the cyan-green ones on Fig. 3.16. To check if different trajectories affect this distribution, we can apply this method to several different cases.

It is worth pointing out now that the reference trajectories considered here are different from the ones shown in Section 3.2 to learn the model parameters, as those were open-loop simulations with only the model, inputting a defined TCP or upper corner trajectory, but these are closed-loop simulations including MPC and feedback, thus following a different numeration. For clarity, the trajectory used in the previous experiments will be labeled as trajectory 0 for this new round, and the new trajectories will be referenced with numbers 1 through 4. All trajectories are different, with 1 being a diagonal line, 2 looping back to the start, 3 having some quick back-and-forth motions and 4 being the one shown in Fig. 2.15 to test the effects of the local cloth base, including a rotation of $2\pi/3$ rad around the global Z-axis. As the absolute tracking errors are different depending on the trajectory, to compare their evolution fairly, we normalize the results obtained between $R/Q = 0$ and 1, as shown in Fig. 3.18.

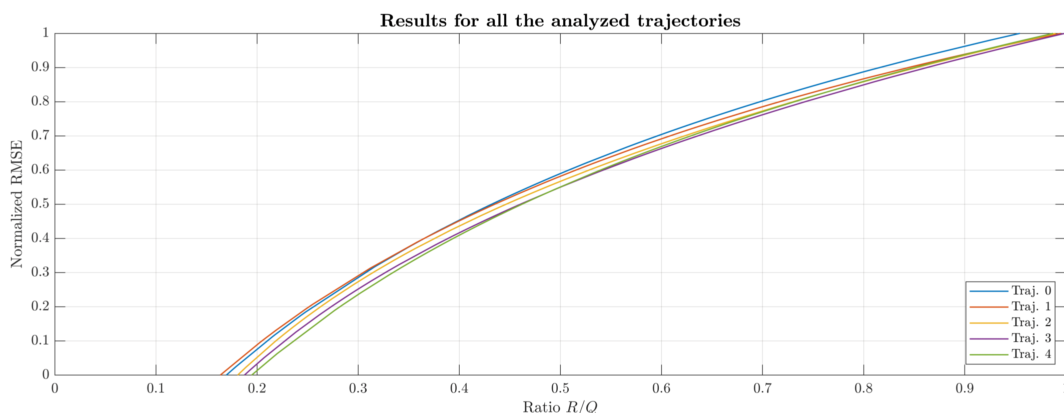

Figure 3.18: Results for all the analyzed trajectories

The dependency of the resulting error with the weights is clear and consistent for all trajectories, so we can set a unique tuning to guarantee the best performance for all of them. Of course, results with high errors obtained with lower R/Q ratios are not shown, but knowing this jump exists, the tuning is now a matter of getting as close as possible to the limit without crossing it for any trajectory, as the exact point does change slightly depending on the case. We must also consider that the absolute difference in error in the shown range does not reach 1 mm even for the worst case, so not being on the very end is not a great sacrifice in performance. A possible combination that works safely for all considered trajectories is $Q = 1,\ R = 0.2$.

Now we could do the same to check if different combinations of T_s and H_p affect the optimal tuning, but having a clear idea of the distribution and zone where the best results are, and the fact that the errors decrease until they suddenly increase significantly, with no local minima in between, we can modify the REPS algorithm to be greedier in its search.

First of all, the initialization will be localized to the zone with $Q = 1$ and low R values, instead of the full range of possibilities. Then, we can set two thresholds, or bounds, to the tested parameters. If a ratio has been tested and the error is high (saturated to the minimum reward, or several times worse than the obtained optimal value), it means we have stepped over to the critical zone, with a value of R too low, and we must not consider any values equal or lower than that one. Conversely, a new optimal reward should narrow the search with values of R greater than the one used to obtain it. This upper bound will not be as strict, because we could be in a situation like the one depicted in Fig. 3.19. If we set the upper bound of R/Q to the best result we obtain in an epoch, it could happen that, as seen in (a1), the best result is past the optimum (orange point inside the blue bounds), but the error has not gone up yet, and it is the best result from the executed samples. If we now bound the search between the previous lower bound (red dot) and the new best sample (orange), as seen in (a2), the optimal weight is no longer inside the search range. To fix this, the upper bound will always be a bit higher than the best result, as seen in (b), so new samples can get progressively closer to the optimum even in these situations (like the green dot). The previous value will always be included with the new samples to ensure that it is kept in case all the samples produce worse results.

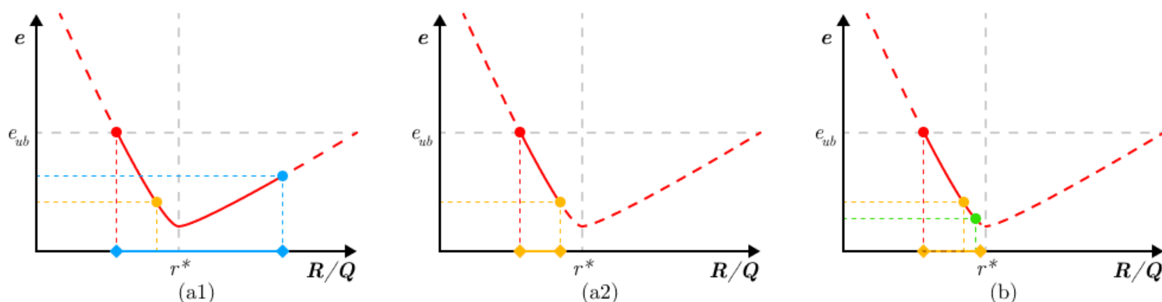

Figure 3.19: Situation where a strict upper bound would leave the optimum out (a), and solution (b)

Of course, in this situation, multiple other learning and optimization algorithms could be used, but with this very slight modifications, we can use the same codes and methods as before.

Given we are looking for a tuning that gives the best tracking performance but can be applied to a wide array of situations, we do not want to learn the exact optimal ratio for one case, but check if the T_s and H_p parameters affect this value in a significant way, and try to find a safe weight combination for all cases without increasing errors over an acceptable margin. Table 3.3 shows all the obtained results for the tested combinations simulating the original trajectory.

Table 3.3: Learnt optimal R/Q ratios for different T_s, H_p

H_p \ T_s	10 ms	15 ms	20 ms	25 ms
15	0.1671	0.1731	0.1651	0.1699
20	0.1669	0.1745	0.1650	0.1699
25	0.1674	0.1746	0.1658	0.1705
30	0.1671	0.1755	0.1655	0.1705

The changes depending on H_p for the same time step are negligible, and even if the differences between the tested T_s are somewhat more noticeable, the range is still contained between 0.1650 and 0.1755. We can safely conclude that the same tuning, with a safe value for all the cases, leads to near-optimal results. For example, executing $T_s = 20$ ms, $H_p = 20$, which had the lowest optimal ratio, with the highest one, the increase in error is about 0.1%. We also have to consider this is only one trajectory, and even if the effects were minimal, some of them had optimal ratios (and more importantly, unstable thresholds) a bit higher. Testing the case that gave the highest ratio, $T_s = 15$ ms, $H_p = 30$, with the trajectory 4, also the one that gave the highest value in the previous tests, we obtain a ratio of 0.1753.

With these results, we can safely say that the tuning of the controller does not depend on the situation, and we can select one combination of weights that yields optimal or near-optimal results for all cases without risking getting too close to the zone where the errors suddenly increase due to R being too low. To guarantee safety, we can choose $Q = 1,\ R = 0.2$, which is a bit higher than all the obtained optimal results, but it is a increase in error of less than 0.5% to account for other possible trajectories and situations that could unstabilize with values of R/Q a bit over what we have obtained.

As a final step before the real implementation, we can simulate the newly tuned controller using the triple model scheme and even adding uncertainties coming from disturbances and sensor noise. This is the scenario we have in simulation which is the closest to the real setup, with a linear COM, an additional linear "backup" SOM (which is actually halfway between being Simulation- and Control-Oriented) to be able to simulate and update the initial states even when feedback is scarce or takes several steps to update, and finally a nonlinear model simulating the real cloth, which is affected by a disturbance and feedback picks up some noise, as it would in a real setup. Using the found optimal structure and weights, with all the other parameters exactly as before, and $T_s = 20$ ms, $H_p = 25$, we obtain the results shown in Fig. 3.20, on both plots to the left. This is clearly not an acceptable situation, but as indicated by the titles of the plots, left together for an easy comparison, the cause of this behavior is actually that we are simulating with a relaxed bound for u of 50 mm per step (about 2.5 m/s of maximum speed), increased

from the necessary limit to speed up the computations. Reducing it back to 5 mm per step, a bit over the maximum slope of the reference trajectory, we obtain the plots shown on the right, which now follow the trajectory as one would expect, and even without needing to introduce stochasticity into the controller.

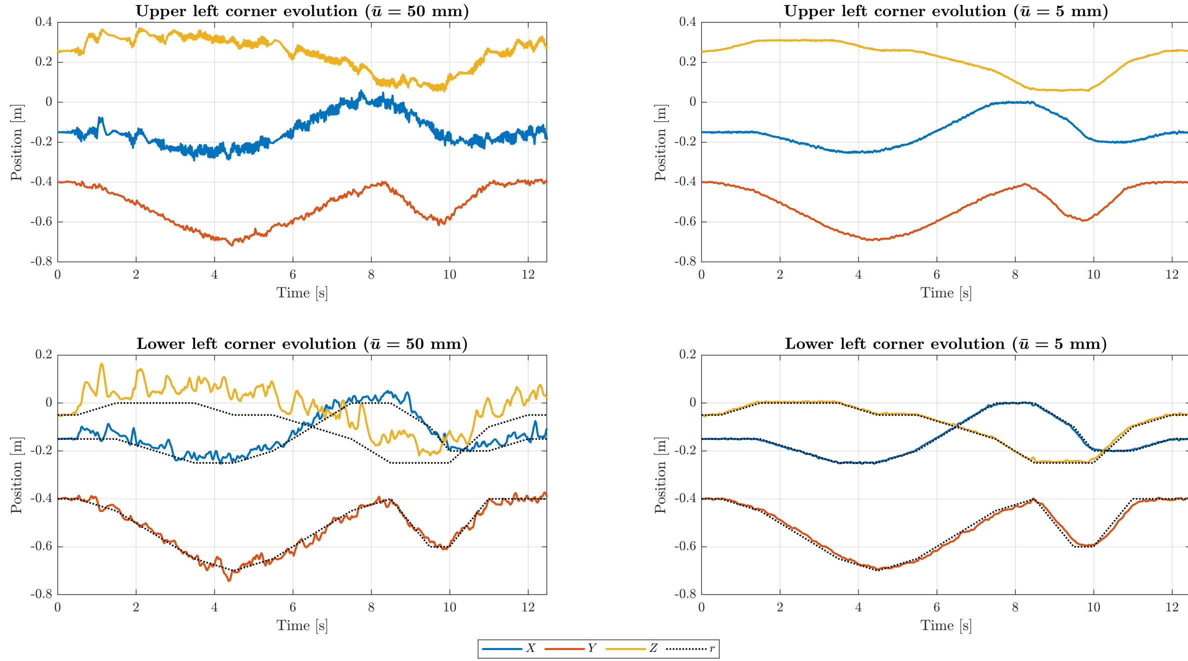

Figure 3.20: Comparison of results using the triple model scheme with disturbances

Of course, for the simulations without noise, this decrease in the bound of u results in the computational time increasing, hence the relaxation done previously, but here it is actually the opposite, as limiting the range of possibilities helps finding an optimum near the initial guess, which is perfect tracking. Given the actual noise and disturbances depend on the real setup and we do not exactly know them, we will not perform learning experiments to find the optimal upper bound of u, as it would certainly change in the real implementation. Instead, a low value will be fixed, and raised only if beneficial. Another option would be to learn online, or during execution, and adapt the corresponding parameters to the current situation, similar to what is done in Adaptive MPC. This is however out of the scope of this Thesis, as we are only considering the applications of RL offline, to improve the MPC before executing.

All in all, we have obtained the best parameters for our MPC, minimizing Δu but without using Q_a, and with simple weights valid for all situations, with the optimal tracking being around $Q = 1,\ R = 0.2$.

4. Real Implementation

One of the main objectives of this Thesis is to implement the control scheme developed, improved and tested in simulation in a real robot. This chapter explains this process, starting with some initial considerations and changes that had to be made to the whole scheme to adapt it to the real scenario, mentioned in Section 4.1. After this, the process to adapt the controller explained throughout the previous chapters is detailed in Section 4.2. The Cartesian controller needed to link our MPC with the robot is described in Section 4.3, while the part relative to feedback data, coming from Computer Vision, is shown in Section 4.4. Once the complete system was working in the real scenario, additional changes had to be done for it to run in real time. These are detailed in Section 4.5.

4.1 Overall Structure and Considerations

The work to implement the cloth manipulation application using MPC and RL in a real robot started early on in the development of this Thesis, and in parallel with the improvements added to the controller and the RL techniques applied to use it in different conditions, and also to tune it automatically. In fact, the origin of some of these changes can be found in problems or constraints discovered while working on the real implementation. Part of this parallel progress has been mentioned in Chapters 2 and 3, where decisions were taken based on what was available or obtained with the real robot.

While the first steps towards a real implementation were centered around adapting the developed MPC codes to be usable with any robot (translating them from Matlab to C++), it was clear that the chosen one would be a 7 Degrees of Freedom (7-DoF) Whole Arm Manipulator (WAM) from Barrett Advanced Robotics, as two were available at the Perception and Manipulation laboratory in the IRI, and previous research regarding cloth manipulation had been tested using this arm. With two robots of the same type being present in the laboratory, this means that further research controlling both upper corners of the cloth independently can be started adapting the work done in this Thesis with some tweaks, but everything would be already tested for that specific robot.

Knowing that the controller would be implemented on a WAM robot also has other more specific advantages, as thanks to the previous research conducted and tested on it, a plethora of codes and functions were available to be used if needed, and ideally, once the MPC system was implemented, it could be connected to different other systems like variations of Cartesian controllers or other learning codes, and vice versa, the now new controller could be used in the future with an even more recent application without the need of adapting a dedicated MPC for it. This meant the integration of the Model Predictive Controller into a library developed by the IRI and specific for the WAM (called `iri_libbarrett`, and adapted from a real-time control library by Barrett Technology), with a unique structure and ways to define inputs, outputs and other variables.

Sadly, not long after starting to look into it, several compatibility issues were found, not just with the newly developed code but also with CasADi in its C++ version. This library, which, as mentioned during the development and redesign of the MPC in Chapter 2, is used to code fast and efficient optimizations, and was chosen also due to its availability in both Matlab and C++, uses some functions only available from C++11 onward, and trying to find older versions had no positive outcome, and neither did trying the opposite, as the older codes in the IRI library could not be compiled with C++11, and splitting the library to compile only the new codes with C++11 was not an option, both technically and practically, as we would not be able to connect the MPC system with any others.

In the end, the implementation on the real setup is done using Robotic Operating System (ROS) in its Kinetic Kame version [45], for Ubuntu 16. ROS is flexible, robust, modular and versatile, meaning the code used for this specific scenario using a WAM can be then applied to any other kind of robot without any effort, as the MPC itself is encapsulated in what is known as a node, and different nodes can be connected easily, even ones coded in different languages (Python and C++ are compatible) and executed from different computers. For the case at hand, using ROS made it easier to connect with the Computer Vision algorithms for cloth detection, as will be detailed in Section 4.4, but it also meant that, to connect to the robot using codes developed at the IRI, a Cartesian controller had to be wrapped as a ROS node, as explained in Section 4.3.

Without going into specific details of how ROS works and all of its possibilities, to understand the final implementation we need to know that there are nodes, which are modules of code that can be run simultaneously, and that they can be connected using messages. These messages, which have their structure and requirements, must to be published by one node onto a topic, and then any other node can subscribe to this topic and receive all the messages published there. Sending commands to the robot is also achieved in a similar way, specifically using what is called an action. The nodes can also be synchronized and all refer to the same common time variable, which is important in real-time applications. Knowing these basics, we now show the full closed-loop scheme of the real implementation using ROS in Fig. 4.1, where each block is a node, which will be all explained one by one in the following sections. The specific topics used to connect them will also be detailed there, to keep this diagram simple.

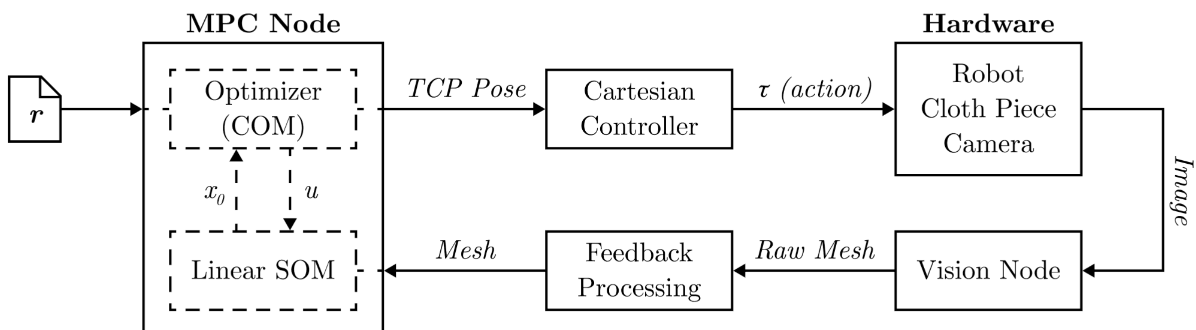

Figure 4.1: Full diagram of the final implementation in ROS

To conclude this section, we show the final situation of the real setup utilized to execute all the experiments. Fig. 4.2 shows one of the two WAMs found in the Perception and Manipulation laboratory at the IRI, the one used in all executions. Additionally, it shows a detail of the end-effector, with a custom 3D-printed plastic adaptor. The TCP is actually inside the central hole, on the metallic plate underneath, as specified by the manufacturer. The plastic part is used to connect with the red one shown on the lower right, which is attached to the rigid link between robot and both upper corners of the cloth. Finally, Fig. 4.3 shows a picture of the complete setup during an experiment, with camera and computer in frame.

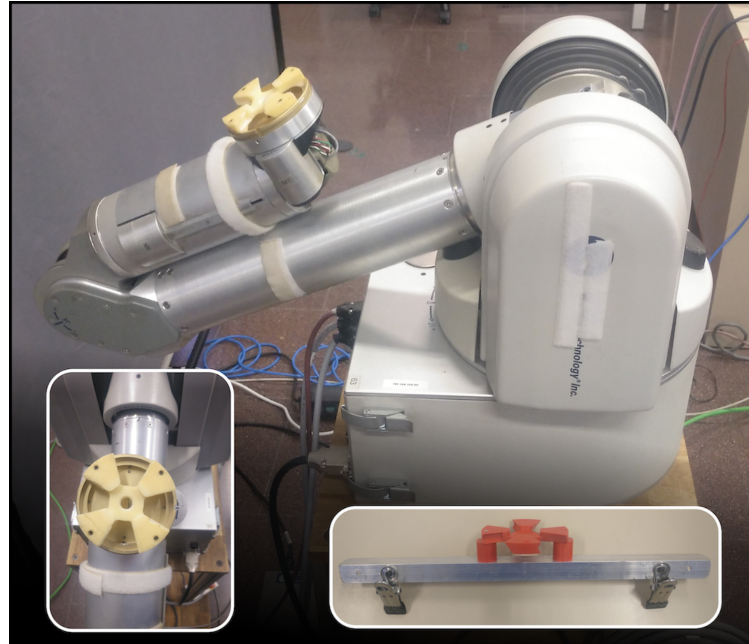

Figure 4.2: Picture of the WAM used in the real setup, and piece that connects to the cloth

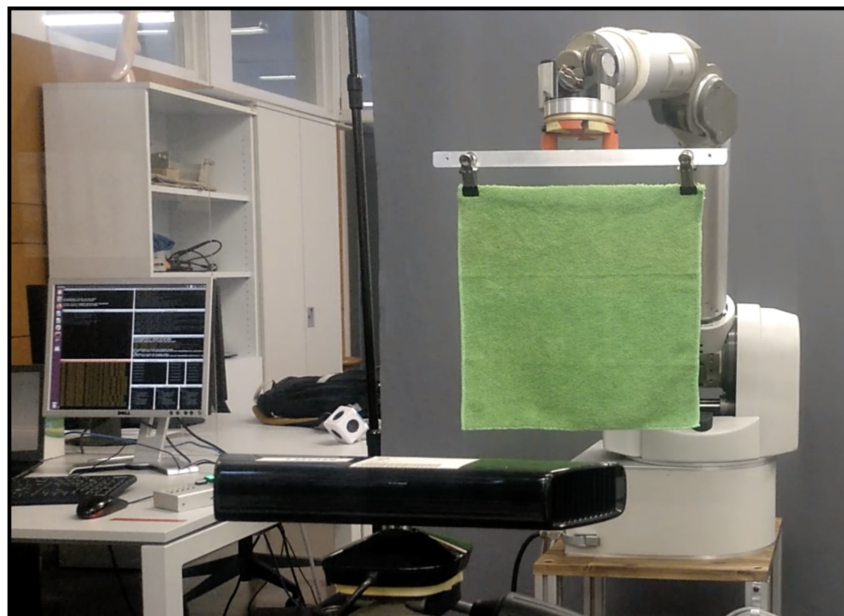

Figure 4.3: Picture of the full real setup during an experiment

4.2 Adapting the Model Predictive Controller

This section is dedicated to explaining the process to transform the designed MPC and existing closed-loop simulation written in Matlab into a code executable in C++, to then adapt this translated code into ROS. Each one of these two steps is explained separately in the following subsections.

4.2.1 Code Translation to C++

From the starting point of the Thesis, it was clear that the codes developed in Matlab would have to be translated. Even while considering using `iri_libbarrett` to connect to the WAM and all other involved systems, this library is written in C++. Once it was clear the final structure would be using ROS, the programming language itself did not change, as C++ is one of the two supported languages, along with Python, with the added bonus of being a compiled language, potentially faster in real executions.

There exist some automatic tools available that convert Matlab programs into C++. However, for codes involving multiple files and functions, they create complex structures of data to communicate between them, which make it harder for a human to understand them and use, tweak or even debug the resulting code, and can also result in higher iteration times with several calls to smaller functions creating other auxiliary variables. The automatic method was discarded quite early on due to these disadvantages, and knowing that the translated code would have to be adapted and easy to debug. Translation was thus done manually, and progressively from a simple function to the whole closed-loop simulation.

Matlab is optimized for mathematical operations, especially vectorized ones, where vectors of different sizes and classes can be used with scalars, concatenated to themselves to grow on consecutive iterations, and other user-friendly applications. To be able to perform all the operations in C++, even after declaring and initializing all variables with fixed size, a specific library was needed. In the end, Eigen was the chosen one, as it allows for a wide variety of matrix operations using simple syntax, and the executions are still fast when the programs are compiled [46].

Besides this library, and as mentioned multiple times throughout this document, CasADi was used to code the optimization problem in C++ too. Given it is the same toolbox as in Matlab, the syntax is similar, and there are tutorials, documentation and examples in all languages [47], easing the translation process. The most notable change, in fact, was having to deal with explicit variable classes and not being able to do operations with different types together, so there is an increase of intermediate steps to change from CasADi symbolic variables to Eigen matrices and vice versa.

Like in Matlab, CasADi is not the optimizer itself, but a tool to connect with one, among several options, so that the users do not need to think about low-level details, and optimizations can be fast even if the problems are formulated using high-level functions. The optimizer used is IPOPT, as in the Matlab code, to keep the same properties and structures already coded, and make the translation process smoother. As mentioned in Section 2.4, this availability in both programming languages was a key feature that lead into using CasADi and IPOPT.

Even before adapting the codes into ROS, it was clear that in the real setup, the codes would run in a computer using Ubuntu 16 as the operating system. This is mentioned now because even though we are using the same tools as before, they needed to be installed in the computer connected with the robot. For the most part, following the installation steps from the CasADi GitHub page was enough, but it is worth mentioning, in case this process has to be done again, or a reader wants to use the developed codes, that to install IPOPT its checkmark had to be manually ticked using cmake GUI (Graphical User Interface) during the installation, besides everything mentioned in the tutorial.

Once all these initial requirements were fulfilled, the translation process began. To achieve a complete closed-loop simulation in C++, not only the main simulation code from Matlab had to be translated, but also all the functions related to initializing and operating with the linear cloth model. This is why not everything was coded into a single file, but several separate function files were made, with their corresponding header files. Even if there were multiple subsequent changes and improvements made to the C++ code, even after adapting it to ROS, making the first translation and all the codes before the final version obsolete (hence why only this final version is attached in the Appendices), the overall structure in different files was maintained, with a file for general purpose functions, like reading from and saving to a CSV (Comma-Separated Values) file, another with functions related to the linear model, and a third one for functions more specific to the MPC, besides the main closed-loop code.

Once reached this point, a closed-loop simulation was available as a standalone C++ code, and the resulting SOM evolutions could be saved and exported to check their correctness. The SOM used in C++ is always a linear model, which in this case was the same as the COM, as the learning experiments from Section 3.2 were not finished at the time. Plotting these results in Matlab, we obtained Fig. 4.4, where we can see how the used reference is correctly tracked. The next step was clear: adapt the code into ROS.

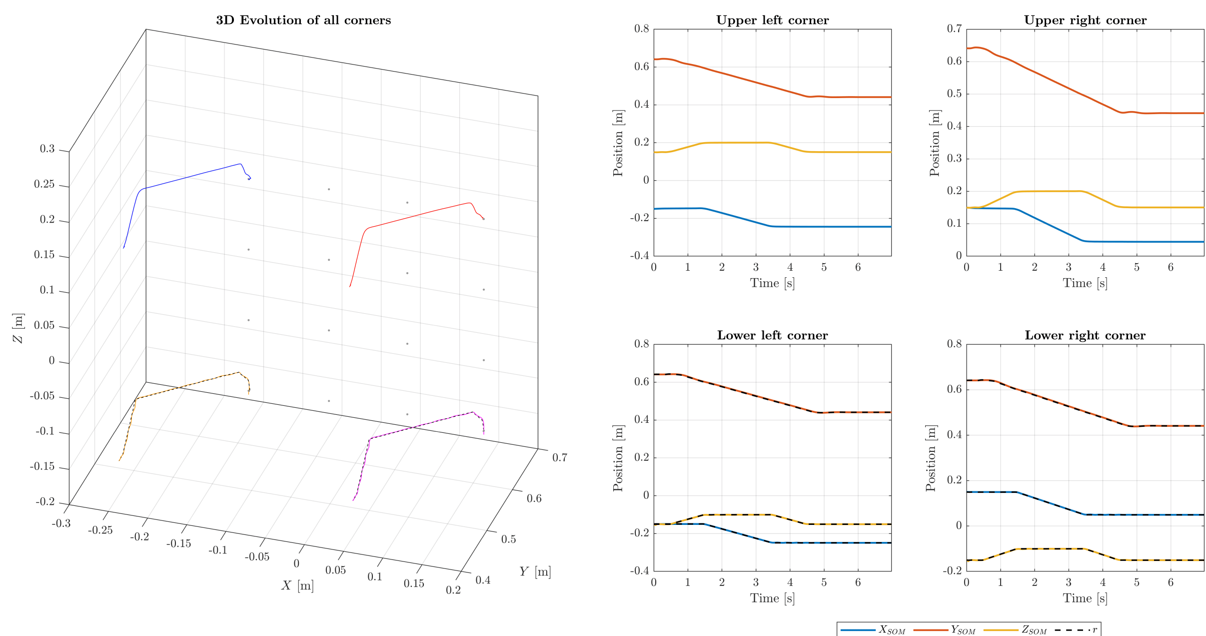

Figure 4.4: Results obtained with the translated closed-loop simulation in C++

4.2.2 Integration in the ROS Structure

The second main step in the process of transforming the MPC into the real scenario is implementing the now translated simulation into ROS. We start with an independent executable and a set of C++ files, and the goal is a node that can be connected in the overall scheme.

For that, we need to define all the necessary inputs and outputs of this node, knowing its contents and place in the diagram. The node itself contains two distinct parts inside: the optimizer, which uses CasADi and has the linear COM inside, and an additional linear SOM. In the simulations, both in Matlab and C++, the SOM is used to represent the real system, but in a real application, that is of course no longer needed. Instead, the linear SOM will be used as a backup, needed to simulate evolutions at a fast rate and give new initial states to the optimizer even when the feedback signals are slow. As will be explained in Section 4.4, this part of the scheme is actually the slowest one, so the linear SOM is essential for the correct performance of the system. Knowing all this, the connections are clear: the MPC node requires the reference trajectory and the most recent state vector captured from the real cloth, and outputs a control signal. In fact, this control signal is not a displacement, as the pure u returned by the optimizer, and not even an absolute position, but a full pose of the TCP at every instant, including its orientation.

In fact, on the final implementation, the MPC node publishes 3 different topics, as seen in Lst. 4.1. This is just a snippet to show the syntax of these definitions, and mention the purpose of each of them. The first one is the absolute positions of the upper corners, i.e., the control signals u added with the previous upper corner positions. This is published with debugging purposes and in case a nonlinear SOM or another node needing only cloth information is connected in the future. The second publisher sends TCP pose messages on its topic. This is the information required by a Cartesian controller to operate, but in the specific setup used, it is not expressed with the right structure, and a third publisher is needed to output Cartesian commands into the topic where the Cartesian controller is subscribed. The second publisher is not removed, once again, for flexibility and debugging purposes, as we can see (or “echo”) these messages manually from a terminal.

Listing 4.1: Definition of publishers and subscribers of the MPC node

```
1  // Define Publishers
2  ros::Publisher pub_usom = rosnh.advertise<mpc_pkg::TwoPoints>
3                            ("mpc_controller/u_SOM",1000);
4  ros::Publisher pub_utcp = rosnh.advertise<geometry_msgs::PoseStamped>
5                            ("mpc_controller/u_TCP",1000);
6  ros::Publisher pub_uwam = rosnh.advertise<cartesian_msgs::CartesianCommand>
7                            ("iri_wam_controller/CartesianControllerNewGoal",1000);
8
9  // Define Subscribers
10 ros::Subscriber sub_somstate = rosnh.subscribe("mpc_controller/state_SOM",
11                                                1000, &somstateReceived);
```

The attached snippet also shows the line needed to define the receiving part of this node, its subscription. In this case, it is a state vector coming from the Vision feedback, when ready, that will update the states of the linear SOM and immediately the initial state of the COM on the next optimizer call.

The reference trajectories of the lower corners are loaded from separate CSV files according to the selected trajectory number. After some initial tests with this number and some other parameters being hard-coded into the source files, they were finally moved into what is known as a launch file, to make changing them significantly easier. All changes done to a .cpp source file (inside the "src" directory) must be compiled, and changing a parameter implied changing the main source code itself. Moving all the variable parameters into a launch file makes it so the compiled code always looks up the values given by the launch file, and these can be changed easily without needing to compile afterwards. The only downside to this is that the nodes can no longer be run on their own, as they always need a launch file to get the parameters from. On the other hand, node parameters can be set through arguments in launch files, which have a default coded value but can be changed on execution on the command line itself, meaning there is not even a need to explicitly change any code at all.

The parameters moved to .launch files (in the "launch" directory, fittingly) are the path where the reference trajectory files can be found, the number of the chosen reference to be followed, COM and SOM side sizes `nCOM`, `nSOM`, the prediction horizon `Hp`, the time step `Ts` and a weight representing the confidence we have in the data received from the Vision feedback, `Wv`, which will be explained in detail in Section 4.4, as even if it is applied inside this node, it is in the callback function called whenever a feedback signal is received from the subscription, and all the operations there are related to the specifics of the Vision-related nodes.

ROS has built-in ways to check time, which is common to all nodes launched under the same master or core, compute durations and iterate at fixed rates. Theoretically, the rate of the execution is defined by T_s, as the frequency is just the inverse of the period. However, in this first implementation in ROS, as also happened in the previous simulations, both in C++ and Matlab, the codes do not run at true real time, as the optimizer is placed in the middle of an iteration, blocking the execution of the subsequent lines of code until a solution is found. If the computational time is lower than T_s, then we can force a sleep time of the remaining time until that value, but if the optimizer takes longer than the theoretical limit, the whole system waits for it to finish. This is of course not an acceptable situation in the final implementation, and will be changed with the modifications explained in Section 4.5. They are left for the end of this chapter instead of explaining them now with the implementation of the MPC node to mirror the progress done during the development of this Thesis. Given how some tests were made without the MPC working at true real time, connecting with the Cartesian controller first, and then closing the loop with processed Vision data before implementing the necessary changes to work in real time, enough data was gathered from each situation, and a comparative analysis is shown in Section 5.1 of the Experimental Results chapter. Without adding this change yet, the MPC node was ready to run, and could also be tested on its own as if it was a simulation thanks to the linear backup SOM, with positive results.

4.3 The Cartesian Controller Nodes

Describing and especially controlling the movements of a robot are not simple tasks, with their own dedicated fields of study. In simple terms, robot motions can be expressed in two completely different ways. The first one is more natural to the manipulator itself, and uses only information coming from its joints, like positions, velocities, accelerations and torques, or what is known as the joint space. Controlling a robot like this does not require of conversions, the robot receives inputs in the joint space and outputs joint information too. However, for humans to plan the movements and understand the output information, it can be difficult to understand, as the correspondence with the natural, Cartesian 3D space is not direct. As a solution to this, we have the second option, expressing the movements in Cartesian space, with positions and orientations that are easily understandable by human users. However, this requires a conversion from Cartesian to joint space to connect with the robot and give it inputs, and then convert the joint information back to Cartesian space to check results or add some kind of feedback.

A Cartesian controller is precisely this intermediate step needed to connect with the robot when the commands being used are expressed in Cartesian space. The process of going from TCP poses to joint values is known as Inverse Kinematics (IK), as the opposite and much easier process of obtaining a Cartesian pose knowing the joint positions is known as Forward Kinematics (FK). It is clear how we need a Cartesian controller between the output of the designed MPC and the WAM, so that the robot can follow the computed control signals. In fact, Cartesian control, besides the necessary IK, can also involve other considerations related to task and motion planning, like collision detection or path finding.

To perform both FK and IK, the dimensions and types of joints are needed for the specific robot being used, as, for example, a joint displacement of a certain angle would move the End-Effector (EE) a different distance depending on the lengths of the links between them. The standard way of expressing these specifications are the Denavit-Hartenberg (DH) parameters [48], a set of four values for each joint that indicate their nature and how they are connected. For the used 7-DoF WAM, the dimensions are public on their website [49], and they were also available at the IRI, given that robot has been used before. The resulting DH parameters are shown in Table 4.1, where the joint variables q_i have been added too, to show that all of them are revolute (affecting the angles θ_i), and none is prismatic.

Table 4.1: DH parameters of the Barrett WAM

Joint	θ_i [rad]	d_i [m]	a_i [m]	α_i [rad]
1	$0+q_1$	0	0	$-\pi/2$
2	$0+q_2$	0	0	$\pi/2$
3	$0+q_3$	0.55	0.045	$-\pi/2$
4	$0+q_4$	0	−0.045	$\pi/2$
5	$0+q_5$	0.30	0	$-\pi/2$
6	$0+q_6$	0	0	$\pi/2$
7	$0+q_7$	0.06	0	0

With these parameters known, we could use a generic Cartesian controller from the literature that works well for our case. Several recent publications have furthered the research on compliant control, important in tasks with human-robot interaction or humans in the workspace [50], while others also apply learning techniques in the Cartesian controllers [51], so there are a lot of viable options. To keep it simple and use work previously developed at the IRI itself, the Cartesian controller developed in [52], [53] will be used. There are several important details about this controller that must be mentioned. First, it is not implemented in ROS, so it must be wrapped in a node to be compatible with the rest of the scheme. Secondly, it includes several versions, from the most simple one with constant gains to a Variable Impedance Control application, also designed for environments with humans nearby. Also, for a trajectory tracking problem, it needs the entire trajectory at once, to process it as a whole and then execute it point by point. Finally, it considers the redundancy present in the WAM to avoid singularities and increase robustness.

This last point has another implication, combined with the fact that, commonly, an IK problem does not have a unique solution, as, in general, the same TCP pose can be reached with multiple combinations of joint positions. For any given pose, this Cartesian controller computes all the possible joint movements that produce no end-effector displacement (null space), and reduces the strength of the joints in those cases, enabling the possibility of an outside force, like a human, moving some parts of the robot (commonly the elbow) without displacing the TCP from the desired position. This is a feature of all the different modes of the controller, including the most basic one using constant gains and no variable impedance.

Wrapping the Cartesian controller into a ROS node is out of the scope of this Thesis, and the process will thus not be detailed in this document, provided that it was carried out at the IRI. While a simple, standalone version with the bare minimum to function could have been coded from scratch without using the described base, it was important that the used controller could be utilized with the WAM at the Perception and Manipulation laboratory without compromising any of the previous work and connections already in place, both using ROS and libbarrett. After some time, the final product was ready to use, adapted to work in real time knowing only the most recent point to follow (updated by subscribing to the corresponding topic) and not the whole trajectory. However, only the version with constant gains was available, separated to be able to test the controller as soon as possible with the minimum needed to test the trajectory tracking application. Luckily, even this version includes the redundancy considerations mentioned earlier. Combining MPC with Variable Impedance Control can be an interesting topic of future research, possible when the full controller is available in ROS.

Besides the Cartesian controller node itself, which is subscribed to a Cartesian goal topic (the one published by the MPC node shown in Lst. 4.1), and uses an action to connect with the WAM, a separate node was created to work in open-loop applications, like gathering data from the cloth mesh to start the learning process of Section 3.2. For these cases, the only required nodes are the Cartesian controller and the Vision feedback ones, but the Cartesian controller needs a specific message, and reference trajectories, even the ones containing just TCP poses, were saved in CSV files with a specific format, compatible with the original version of the controller that loaded the whole trajectory at once, but not in ROS.

This new node, dubbed "Read Node" due to its main function, first loads the trajectory contained in a given CSV file with the following format: one point on each row, time steps on the first column, then 7 columns corresponding to joint positions (of which only the first one was used to initialize the robot on the initial Cartesian controller, and now is not considered), 3 more columns for the positions of the TCP in X, Y and Z, and finally 4 columns corresponding to the orientation expressed as a quaternion, with the scalar component first. Once the file is processed and the Cartesian trajectory is fully saved on a variable, the node enters a loop with a rate defined by the time steps of the first column of the input file, where it picks the next point in the trajectory, formats it as a Cartesian command message, and publishes it on the topic the Cartesian controller is subscribed to. The structure and nodes involved in an open-loop execution like gathering cloth evolution data is shown in Fig. 4.5.

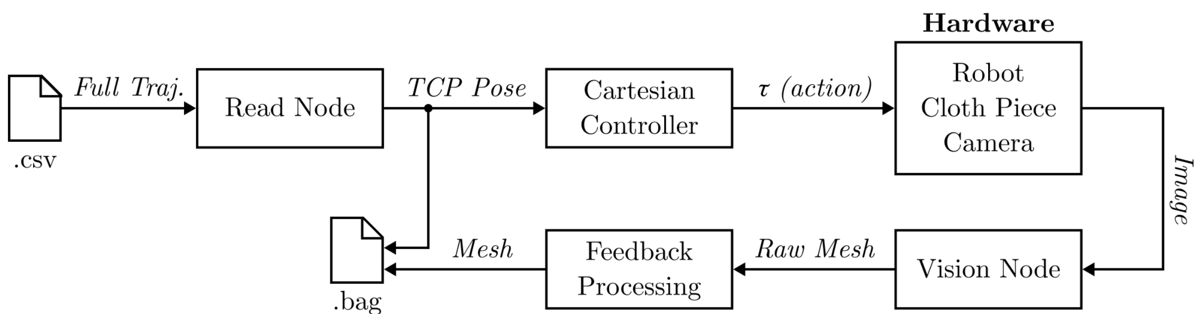

Figure 4.5: Diagram of an open-loop implementation in ROS using the Read Node

In this case, instead of closing the loop, the data coming from the Vision nodes is saved together with the input commands into a rosbag file, directly capturing the messages being published in the specified topics, which allows for a reproduction of them afterwards to simulate data being published without using the real setup, and of course they can be converted to TXT or CSV files (one per topic) to later process the data using Matlab or any other software.

The main code for this Read node can be found following the link to the full source code attached in the Appendices document. Additionally, a complete guide is included both to use the wrapped Cartesian controller, initializing the action and sending goals via terminal, and to edit it, for example, to change the gains, as the process has multiple steps in different directories that must be done every time.

4.4 The Vision Feedback Nodes

The final section of the full scheme to be described is the one including the feedback nodes, with the steps required to go from a camera back into the MPC. This section is divided in three parts, corresponding to distinct nodes needed to perform these steps and to enable an actual closed-loop implementation. The first step is of course capturing the data and applying the required Computer Vision algorithms to identify the cloth and output a mesh. This is described in Subsection 4.4.1. The positions of the nodes of this mesh are given in the coordinate frame of the camera, so a calibration process is needed, as explained in Subsection 4.4.2, to know how to perform a change of base. Finally, Subsection 4.4.3 details all the changes necessary to close the loop and update the initial state of the MPC using real feedback data.

4.4.1 Obtaining the Cloth Mesh

Converting image data given by a camera into a cloth mesh where we know the positions of all the nodes is not an easy task, let alone converting an image to the required state vector. Processing images and their data is part of a whole dedicated field known as Computer Vision, and the algorithms required to extract the required information, identify the cloth piece from an arbitrary picture to then build a mesh according to the detected shape and output are far beyond the scope of this Thesis. Of course, a code that performs these tasks is required to close the loop with real data, but instead of developing one from scratch, we can use an already existing ROS node.

The perfect candidate is the Real-time multi-cloth point cloud segmentation ROS package [54], developed by M. Arduengo et al. at the IRI itself to detect cloth pieces, separate multiple ones by color, and finally merge points that are close both in distance and color into point clouds representing the cloth pieces. The ROS node is coded in Python, but as mentioned in Section 4.1, both languages can be used in the same structure or full scheme including multiple nodes, the only requirement is to have a working connection, for example, a subscriber in a node written in C++ receiving messages published by this node on a certain topic.

An important note about this node is that it uses a You Look Only Once (YOLO) object detection algorithm with a neural network which is resource intensive and needs a powerful GPU, not found in the computer connected to the WAM at the laboratory. This is however not a problem when using ROS, as several different computers can be connected under the same core or master, and launch nodes from different machines onto the same structure to work together. This Cloth Segmentation node can be launched in one computer with the camera connected to it, publish the mesh data into a topic, and the node subscribed to this information can be in another computer, the one actually connected to the WAM. While the node publishes several topics, the one needed for our application is `cloth_segmentation/cloth_mesh`, containing the positions of all the nodes of the obtained mesh. To compute these positions in the three axes of space, the camera needs to output a Red-Green-Blue-Depth (RGB-D) image, and the one used for the real setup is a Kinect camera like the one shown in Fig. 4.6. These positions are expressed in a local base relative to the camera, and must be converted to the global reference frame to be able to update the initial state of the MPC. To do that, first we need to know where the camera is located, with the calibration process described in the next subsection.

Figure 4.6: A Kinect camera, used to capture RGB-D images and obtain current mesh positions

4.4.2 Robot-Camera Calibration

All data captured by the camera is naturally expressed in its own coordinate frame, relative to its position. Given the images are RGB-D, with a depth component, the location data are full 3D positions, with the XY-plane being defined as usual in Computer Vision and graphics, with X across to the right and Y down, and the additional depth assigned to the Z axis, all of them from the point of view of the camera, of course.

The Cloth Segmentation node outputs the positions for all mesh points, but they are still in this local "Camera" base, and thus require a transformation to be used with the states considered by the rest of the nodes, which are in "Robot" or "World" base. To be strict and consider all the details involving coordinate frames, the MPC node has a change of base inside to call the optimizer in a local "Cloth" base, as explained in Subsection 2.5.2, but this is contained within the same node, and both the inputs and outputs are represented in the global base. The required transform only needs the position and orientation of the camera expressed in global coordinates, which remains constant during the execution of all experiments unless the tripod where the camera stands is moved by an outside force, so an option would be to measure the distance between robot and camera and align the two as accurately as possible. Another option would be to use a calibration sheet, with distinct bright colors and high contrast, in a fixed known position to locate the camera checking where it detects the sheet. However, a third option can be implemented using only data and material required for the experiments themselves and nothing more. For it, we need a third intermediate coordinate frame, the End-Effector one. A schematic version of this setup with the three frames is shown in Fig. 4.7.

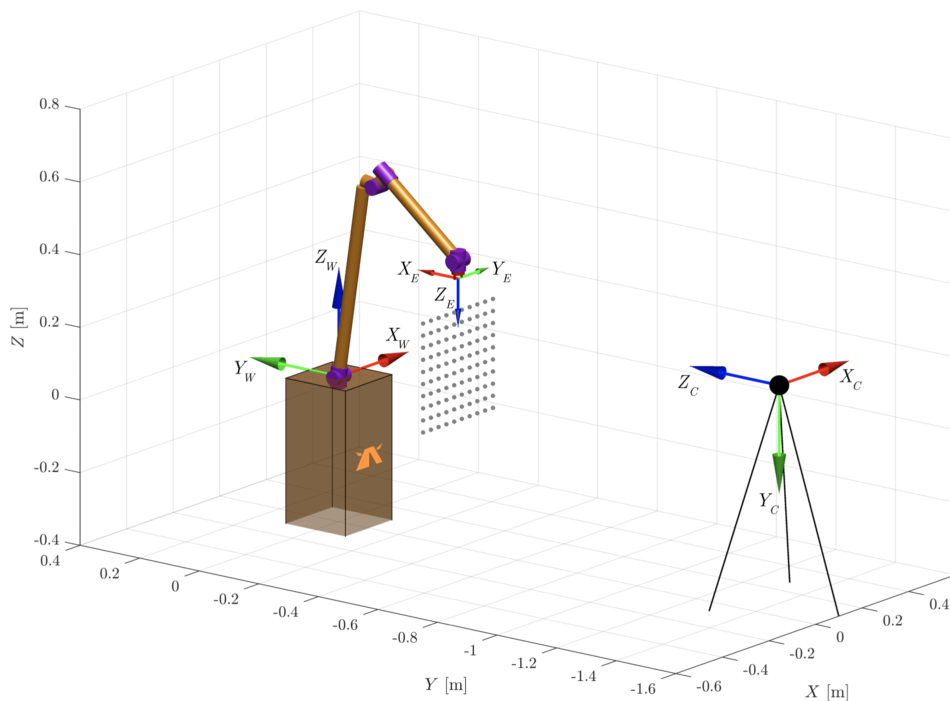

Figure 4.7: Diagram of the setup with World, End-Effector and Camera coordinate frames shown

The idea is that, with a cloth piece attached to the robot, both independent frames, camera and world/robot, can know where this intermediate one is. For the WAM, it is a straightforward task, as the EE base is obtained just applying FK, and using ROS, this is a message being published by default already. Once launched and ready, it can be found in `iri_wam_controller/libbarrett_link_tcp`, and we can subscribe to it from any node. For the camera, the operation involves a couple of steps and requires the cloth, as it is what the Computer Vision algorithms in the used node detect and output a mesh of. Therefore, we can obtain something similar to the local cloth base used in the MPC, and then move it an offset to the TCP, as the upper corners of the cloth and the robot are linked together with a rigid piece.

In this case, we know that the Z-axis of the EE base, Z_E, always points away from the robot into the cloth, with Y_E being parallel to the axis that connects both upper corners of the cloth (it goes along the rigid link), and X_E being defined by the cross product of the other two. If the nodes of the mesh published by the Vision node are ordered in a consistent way, and they must be to be able to use them as a new state vector when closing the loop, we can subtract the position of the top right corner (last node in the usual left to right, bottom to top numeration used across this Thesis) minus that of the top left corner to obtain the direction between them, in other words, the direction of the Y_E axis expressed in camera coordinates. We can do the same with a node in the bottom minus the position of the corresponding top one to obtain the direction of Z_E in camera base. While the top corners are connected by a rigid link and the direction between them will always be the same as Y_E, this is not true for the bottom nodes in a general case. To reduce the possible errors derived from this, we can subtract positions of nodes of the two uppermost rows, and ensure the calibration process is always done with the cloth extended straight down, like in the situation shown in Fig. 4.7. With two axis defined, we have the whole EE base expressed in camera coordinates applying $X_E^C = Y_E^C \times Z_E^C$.

While the previous cross product is true with the axis expressed in both World or Camera frames, it is a good moment to introduce the notation commonly used for rotations, transforms and base changes, given how a single subscript does not suffice: we want to express the X-axis of the EE base, but expressed in camera coordinates, not relative to the world base. This is why we add a superscript indicating "expressed in this coordinate frame". This can be applied to axes, points in space, rotation matrices and transform matrices, for example $R_E^C = \begin{bmatrix} X_E^C & Y_E^C & Z_E^C \end{bmatrix}$ if each axis is a column vector. It is also worth noting that the opposite rotation, camera base expressed in EE base, can be obtained with the inverse of this matrix, which in orthonormal bases is also the transpose: $R_C^E = \left(R_E^C\right)^{-1} = \left(R_E^C\right)^T$.

With the rotation matrix found, the last step is to find the origin of the EE base expressed in camera coordinates. This would be direct if the Vision node detected the TCP directly, but we can find a simple workaround with the data available by finding the position of the central node of the top row of the cloth mesh and adding the known constant offset from the cloth to the robot. To be more robust against noise, instead of simply using the exact node in the middle (or the two central ones if the mesh side size n is even), we can average the positions of all the nodes in the top row to find its center. Once this is done, we can add the offset Δh in the negative Z_E^C direction, to get the TCP expressed in camera coordinates, p_E^C, using only cloth data.

A 4×4 transform matrix T includes all the information necessary to change from one coordinate frame to another, and is formed by a rotation matrix, a point vector and a scale factor, which in robotics is always 1. These matrices are defined as seen in (4.1) for any two bases A, B.

$$T_A^B = \begin{bmatrix} R_A^B & p_A^B \\ 0 & 1 \end{bmatrix} \tag{4.1}$$

These matrices can also be inverted to obtain the opposite transformation, and chained together with other matrices to obtain direct transforms between bases using intermediate steps. This is exactly what we will use to obtain the camera base expressed in world coordinates, as now we have obtained T_E^C, and T_E^W is the product of applying FK. This process is detailed in (4.2).

$$T_C^W = T_E^W \cdot T_C^E = T_E^W \cdot \left(T_E^C\right)^{-1} \tag{4.2}$$

As this equation implies, the exact position of the EE base does not matter, as both transforms will change accordingly to maintain a constant relation between the two fixed reference frames.

This is the idea and process behind the “Calibration” node, which must be launched after preparing the WAM and camera but before any closed-loop execution. It subscribes to the two required topics, cloth mesh and TCP pose, and publishes the resulting camera pose (with position and quaternion instead of matrix T, to use a standard ROS geometry message) in `/mpc_controller/camera_pose`.

Theoretically, the calibration process would be done the instant we have data from both sides, and it could be done regardless of situation, even during movement. In practice, we cannot calibrate during movement or try to improve the calibration during execution without completely changing the codes to consider that the received messages from each subscription come at arbitrary times and different rates, so the TCP data might have updated but the latest mesh data is still from before moving. This is the reason behind the calibration process being done before starting the experiments. Additionally, the output data is very noisy, and we cannot blindly trust a single point received from the Vision node. This is why the calibration process is kept running for a short period of time, fixed at 20 s experimentally, and the T_E^C transforms are not updated directly to match the new received data, but are averaged during this time every time a new mesh is received, using the update rule shown in (4.3), where N_i is the total number of meshes received, and $(T_E^C)_i$ is the newest one.

$$T_E^C \leftarrow \frac{(N_i - 1) \cdot T_E^C + (T_E^C)_i}{N_i} \tag{4.3}$$

Finally, the T_C^W transforms published by the calibration node must be received by another node and the last value must be saved for the execution of all experiments. In the end, a launch file was added to run both calibration and feedback processing nodes, and then kill the calibration node after 20 s. This way, the processing node, which is subscribed to the camera pose, as explained in the following subsection, will save the last valid calibration message internally, as it will keep running during all executions.

4.4.3 Closing the Loop

Even with all the positions of the cloth mesh known and being published into a topic, and having a calibrated transform to express them in the same frame as the backup SOM states, there are still some steps needed to finally connect this data and obtain a full closed-loop scheme. They are divided between a dedicated feedback processing node and the callback function executed on the MPC node whenever a new state vector is received.

First of all, the processing node is subscribed to the mesh positions published by the Vision (cloth segmentation) node. We need the nodes inside the mesh to be ordered in a consistent way, so the state vector can be built automatically with the captured data. Actually, this is not always the case with the used node, which sometimes outputs the nodes in a different order, going by columns instead of by rows, starting at the top instead of the bottom, or other variations which are not the required left to right, bottom to top. This is the first operation done in the processing node, to ensure, for example, that the first n nodes are on the same row with increasing X_C. At this point, we must consider that with a static camera and a moving cloth piece, this ordering has its limitations when the cloth rotates along the Z_C axis more than 45 degrees, when the "top row", including the corners being controlled, would be more vertical than horizontal, as seen by the camera. Other limitations come with the range of vision of the camera itself, which needs to see the majority of the cloth frontally to identify it and extract its mesh. For the experiments carried out in this Thesis, this limitation was considered and movements were kept in an acceptable range so the Segmentation node could always process the Vision data correctly.

Of course, this step must be done to all the nodes using data published by the Vision node, to ensure the ordering is correct. This is why this step is in fact not only done in the processing node, but also in the calibration one, whenever a new mesh is received.

When the processing node is launched together with calibration, it subscribes to the camera pose topic and receives successive transforms of the camera frame expressed in world frame, T_C^W. Once the calibration node is terminated (done automatically with a launch file), the last received transform is kept as the value to use for all the following executions, until the processing node is stopped too. When a mesh is received and ordered, all three coordinates of the same node i are expressed as a position vector in the camera frame p_i^C, to then obtain its position in the global base with $p_i^W = T_C^W p_i^C$. The first half of a full state vector is obtained reorganizing these positions. The second half, with velocities, is computed as the difference in position between two consecutive points over the time elapsed between them.

With the state vector completely constructed, it could be published directly and used by the MPC node. However, given the noisy nature of the captured data, a filtering process was added before sending the state vector to the MPC node. When filtering gathered data offline, we can use both previous and posterior time steps to apply filters and smooth the resulting trajectory, as we have the complete evolution from beginning to end. In an online application like this one, the future points are unknown, therefore, if a filter uses both previous and following points to filter the central one, it needs to wait an additional time interval to output the filtered result, delaying the feedback signal.

This delay is thus related with the rate at which new mesh data is published, and would not be significant if this rate was several times faster than the MPC sampling frequency. Unfortunately, in the real scenario, this is not the case, with the Vision node being the slowest one, publishing data around every 100 ms (not an exact rate, data is published when processed), when the considered time steps for the MPC are between 10 and 25 ms. This means adding four to ten steps of delay, thus sending this data to the MPC as if it was from the current step is completely unreasonable.

There exist other filtering options using only current and past data points, such as moving average filters [55]. The most common variants are shown in (4.4), using three data points: FMA is the “Future” Moving Average, using the previous and following data points, SMA is a Simple Moving Average (arithmetic mean), WMA is the Weighted variant, and EMA is the Exponential one, which is computed iteratively.

$$FMA \rightarrow y_f(k) = \big(wy(k+1) + y(k) + wy(k-1)\big)/(1+2w) \tag{4.4a}$$

$$SMA \rightarrow y_f(k) = \big(y(k) + y(k-1) + y(k-2)\big)/3 \tag{4.4b}$$

$$WMA \rightarrow y_f(k) = \big(w_0 y(k) + w_1 y(k-1) + w_2 y(k-2)\big)/(w_0 + w_1 + w_2) \tag{4.4c}$$

$$EMA \rightarrow y_f(k) = \alpha y(k) + (1-\alpha) y_f(k-1) \tag{4.4d}$$

The disadvantage of using these filters is that their output corresponds to an average of the last several points, which in an actual movement, means the filtered output lags behind the real position of the mesh, pulled back by the previous positions. This is shown with a simple example in Fig. 4.8, comparing all the considered filters using three data points and $w = 0.6$, $w_0 = 1, w_1 = 0.5, w_2 = 0.25$, $\alpha = 0.75$.

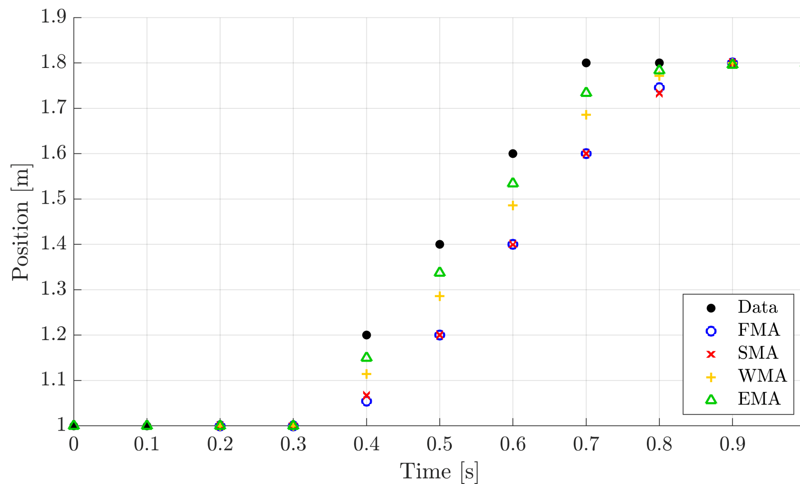

Figure 4.8: Comparison between the different considered filters in a scenario without noise

We can see how waiting a full time step to get the following point to publish a filtered version of the previous one is, together with SMA, the worst option in terms of the added lag in a movement without noise. While moving, these two options output the exact location of the real data with a whole step of delay. With both WMA and EMA, this effect can be reduced significantly depending on the values for the weights. For all these filters, there is always a balancing act between considering other data points to filter noise better, and virtually adding more delay.

With this in mind, all these options were coded into the processing node to test which one resulted in the best performance experimentally. It is clear that regardless of the chosen one, there is going to be an added delay which will have to be accounted for before finally closing the loop and using the captured data as the new initial state for the MPC. Even without using any kind of filter, the feedback signals have the slowest rate in the entire scheme, meaning not only that several iterations of the controller will use the same sample as the most recent one, but also that when the feedback data reaches the MPC node, it might already be the output corresponding to a few steps before. Knowing this, the steps required to process the feedback information and obtain a closed loop will be explained in this chapter, and the selection of the filter itself, which needs these operations in place to compare the performance of each one, will be shown in the Experimental Results chapter, in Section 5.2.

To account for the time elapsed from the moment of capturing the data until it is received by the MPC node, we can use the time stamps on the headers of ROS messages. Instead of sending the message with a time corresponding to the moment of sending each message, we can keep the time of the original captured data given by the camera. Once the mesh data is ordered, in the correct base and filtered, the processing node publishes the state vector into `mpc_controller/state_SOM`.

Inside the MPC node, a callback function is triggered each time a new message is received in this topic. In this function, the message is decoded to obtain the state vector and the time stamp indicating the moment this data was captured. The delay can be computed accessing the current time with `ros::Time::now()` and subtracting this stamp. The first step is comparing the feedback and SOM mesh sizes, and reducing the former to have the same size as the latter if necessary. The following step consists in updating this data to current time simulating the evolution from its acquisition until current time. To do this, a linear cloth model identical to the backup SOM is used (same A, B, f_{ct} matrices) and we need a new variable saving the history of the past control signals obtained by the MPC.

This variable is, of course, a finite concatenation of vectors, therefore, only a limited number of steps can be simulated to update the feedback data. Knowing the Vision node outputs data around every 100 ms, and that the fastest time step considered for the MPC is 10 ms, the maximum number of update steps was set to 12, to add a small margin to the required 10 steps in the worst case scenario, knowing the filtering process will increase the delay already present due to the slow rate of the feedback signal.

If the interval between acquisition and current time is longer than the number of steps times T_s, the feedback signal is discarded immediately, as the delay is too large, meaning the data is too far into the past to be applied as a new initial state. When this is not the case and the state vector is updated into current time, then it is compared with the current states of the backup SOM being simulated internally on the MPC node. If the distance is also over a certain threshold (set empirically to an average of 30 mm for all nodes), the feedback data is also discarded, as it was observed that, even after filtering, some spurious signals with spikes of noise still made it through (albeit with a reduced difference), and this last safeguard was needed to avoid updating the initial state for the MPC and also the current backup SOM state to an incorrect value. With the backup SOM keeping the states updated, even after a discarded sample, the following one can be applied if it satisfies the previous criteria.

Finally, after updating the feedback data to current time and filtering distant results, the backup SOM state vector is updated to the received one, and the initial state of the COM that goes into the optimizer is obtained with the updated value too. However, the received data still maintains some variance added from the noise and cloth detection processes, and trusting it blindly updating the state vector directly resulted in nervous evolutions and worse errors. This is why a final parameter, a Vision weight W_V, was added, to consider the reliability of the feedback data versus the state vector obtained simulating the backup SOM. This way, the update that finally closes the control loop follows (4.5).

$$x_{SOM} \leftarrow W_V x_{Vision} + (1 - W_V) x_{SOM} \tag{4.5}$$

The effects of this weight, together with the sampling time T_s and the prediction horizon H_p, are analyzed using experimental data in Section 5.3. As a summary of all the steps the captured Vision data must go through to close the loop, we now show Algorithm 3.

Algorithm 3 Steps to close the loop with Vision feedback data

Require: Camera publishing RGB-D images

1: **for each** image captured at time t_c **do**
2: Use the Cloth Segmentation node to get the mesh positions $p(t_c)$
3: Apply a filtering process: FMA, SMA, WMA or EMA
4: Order the node positions from left to right, bottom to top
5: Apply a base change from camera to world reference
6: Publish mesh using original acquisition time stamp, $x_V(t_c)$
7: Receive mesh on MPC node: callback function
8: Compute delay $\Delta t = t - t_c$
9: **if** $\Delta t > \Delta t_{max}$ **then**
10: Discard feedback data
11: Exit callback function
12: **else**
13: Update data simulating $\lceil \Delta t / T_s \rceil$ steps, obtain $x_V(t)$
14: **if** $\left\| x_V(t) - x_{SOM}(t) \right\| > \Delta d_{max}$ **then**
15: Discard feedback data
16: Exit callback function
17: **else**
18: $x_{SOM}(t) \leftarrow W_V x_V + (1 - W_V) x_{SOM}$ ▷ New initial state for the MPC
19: **end if**
20: **end if**
21: Exit callback function
22: **end for**

4.5 Implementation in Real Time

Until now, all the closed-loop implementations had the optimizer as a blocking step in the MPC, both in simulation and in the MPC node. This was not a problem in simulation, as after any arbitrary time needed by the optimizer to find the solution, we can force a step of T_s to simulate the SOM, get the feedback signal and continue with the next iteration. Even if each iteration takes a different amount of time to complete, the evolutions are virtually simulated at constant step times. This is not the case in a real implementation, where the optimization time is added to the computational time of all other commands in an iteration of the MPC node, and if the total time is greater than T_s, each step cannot be completed in time, causing the output control signals to be delayed, and the incoming feedback signals to be incorporated later too.

This is why we need to change the structure of the MPC node to avoid having the optimizer as a blocking step, and ensure each iteration is completed within T_s and there is always an updated control signal being outputted at a constant rate. The simplest way to have two codes running in parallel and not blocking each other in ROS is using two separate nodes. This is the main change to the overall ROS structure needed for a real time implementation, and thus a new diagram is shown in Fig. 4.9. The old MPC node is now divided into a node dedicated only to the optimization problem (Opti Node) and another one (RT Node) with just the linear SOM that ensures a constant output rate of TCP poses, one every T_s, and also handles the feedback signals.

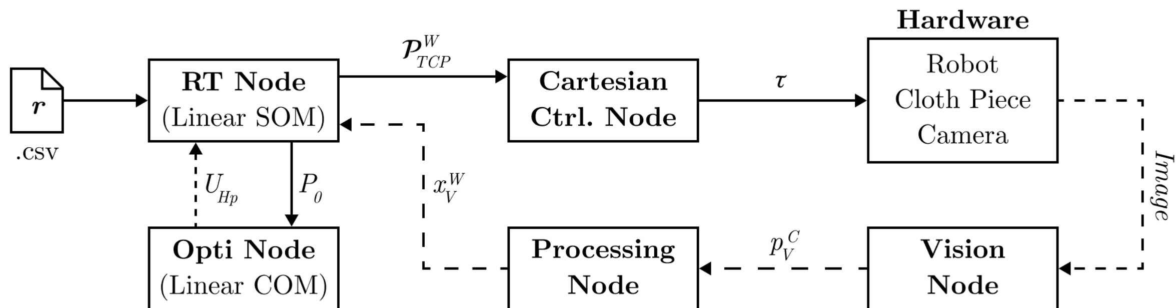

Figure 4.9: Diagram of the ROS implementation in real time

In this diagram, we have marked with a solid line the messages that are published at a fixed rate, one every T_s, and with dashed lines all the others that are not. The separation between dashes is proportional to how slow these messages are published, with the optimizer being the fastest among them (sometimes will finish before a full time step, sometimes it will take longer, depending on initial conditions and reference). The camera outputs RGB-D images at 30 Hz (33.3 ms between frames), but the Vision node needs around 100 ms to process this data and publish the mesh positions, thus its output and the output of the processing node are the slowest ones. Additionally, the contents of each message of the nodes that have not changed now are indicated using the notation described in Section 4.4, with p^C_V being the positions of all nodes expressed relative to the reference frame of the camera, x^W_V being the state vector obtained with Vision data in world base, and $\mathcal{P}^W_{TCP}$ is the pose of the TCP in global coordinates (same information as a transform matrix T, but expressed as position and quaternion).

The new RT Node keeps the same callback function to process the feedback data and update the SOM state. In each iteration, at a constant rate $1/T_s$, right after checking for new feedback messages, the initial state for the optimizer is extracted from the updated SOM. This state is joined together with the updated horizon for the reference and the previous applied control signal in a new variable of initial parameters, P_0. This is what the RT node publishes with a new custom message to update the initial conditions of the optimization problem in the Opti Node.

The optimization node processes this data in a callback function and starts optimizing. While the theoretical rate is also set to $1/T_s$, the optimization can take longer than that to complete. Once it is done, instead of publishing the first control input to apply, $u(k = 0)$, the full sequence of control inputs from $k = 0$ to $k = H_p$ is saved in a new variable, U_{Hp}, and published for the RT node to receive. This is done precisely due to the fact that an optimization might take more than a time step to complete, and thus the RT Node might go through multiple iterations before the Opti Node publishes an updated control input. This way, instead of applying the same control input several times until the next one is obtained, which would be no improvement over publishing the new ones only once, we use consecutive predictions in the horizon and keep the backup SOM updated in real time.

With just these changes, a problem still persists with the optimizer node. The published control signals allow it to take more than a T_s to complete an optimization if needed, but the published concatenation of control vectors is finite, with H_p steps as an absolute maximum time before a new sequence is required by the RT node, and there is no imposed limit to how long an optimization can take. For extreme cases in the real setup, with noise, a demanding trajectory and a noisy initial state, if not stopped, an optimization can take even seconds to complete, when the maximum prediction time ($T_s \cdot H_p$) used is 750 ms, and in the majority of experiments it is around half a second.

The simplest solution to avoid the solver getting stuck and taking more time than allowed is adding a maximum optimization time, or a timeout, so that if this time is reached, the solver stops regardless of its situation and progress of finding the optimal solution. Then the optimizer node can look for updated initial conditions, and restart the optimization with these new parameters, hopefully finding a solution before the maximum time. The easiest way of implementing a timeout in the developed code is to add a `max_cpu_time` termination condition to IPOPT, the solver used in conjunction with CasADi. This time threshold must be lower than the total prediction time, and even lower than half this time to ensure that at least two optimizations have been tried before the end of the horizon. In the end, the maximum time was set to $T_s H_p/4$ empirically, as it was enough to complete optimizations regularly, and only produced timeouts in scenarios where the initial parameters were demanding and the optimization would have taken seconds. With new parameters being received every T_s, new ones are always used on retries, and consecutive timeouts were only obtained in situations where the cloth had orientations where the mesh was hard to compute by the Vision node, in very fast movements, and for extreme combinations of low T_s and high H_p, for example 30 steps or more at 10 ms.

As a final remark for this chapter, we must reiterate that the changes needed for a real-time implementation have been left for the final section to be loyal to the order in which the changes were implemented in the real setup, and because, with gathered data of all the steps (no feedback, feedback without real-time and everything included), we can show and compare the results in the following chapter, concretely in Section 5.1, to show the importance of each modification. Additionally, no experimental results using the final real setup have been shown in this chapter to clearly separate its description and implementation (the overall ROS structure, all the nodes and necessary changes to obtain its definitive version) from the experiments carried out in this setup, the results obtained and the analyses derived from them, which deserve their own dedicated chapter.

5. Experimental Results

Once the full closed-loop control scheme was implemented in ROS to work with a real setup, experiments could be done to test each modification and analyze different alternatives to find the one that yields a better reference tracking. This chapter contains the outcome of these experiments, starting with Section 5.1, where the differences between having a blocking optimizer and working in real time are shown. After the real setup is actually running in real time, a filter was needed for the Vision feedback. The selection of the most suitable one is shown in Section 5.2. The new closed-loop scheme works with multiple values for the control parameters, namely T_s, H_p and W_V, that can affect the tracking performance. An analysis to find the best combination is shown in Section 5.3. Finally, the system was tested in demanding situations, with fast trajectories, rotations and disturbances, as seen in Section 5.4.

5.1 Effects of Working in Real Time

The implementation in the real setup was progressive, and the changes to work in real time were added after testing the MPC node by itself and after closing the loop with real Vision feedback. Each change was tested, and data was gathered, meaning we can clearly show the effects of each one step by step.

The first results, shown in Fig. 5.1, correspond to the MPC node working without real feedback and with the optimizer as a blocking step in the iterations. In the lower left corner, the trajectory at a constant rate is shown, in this case $T_s = 0.025$ s, finishing after 22.5 s. Clearly, the results are extended in time, and in the lower right corner, the reference has been stretched 2.72 times to reach the end of the execution.

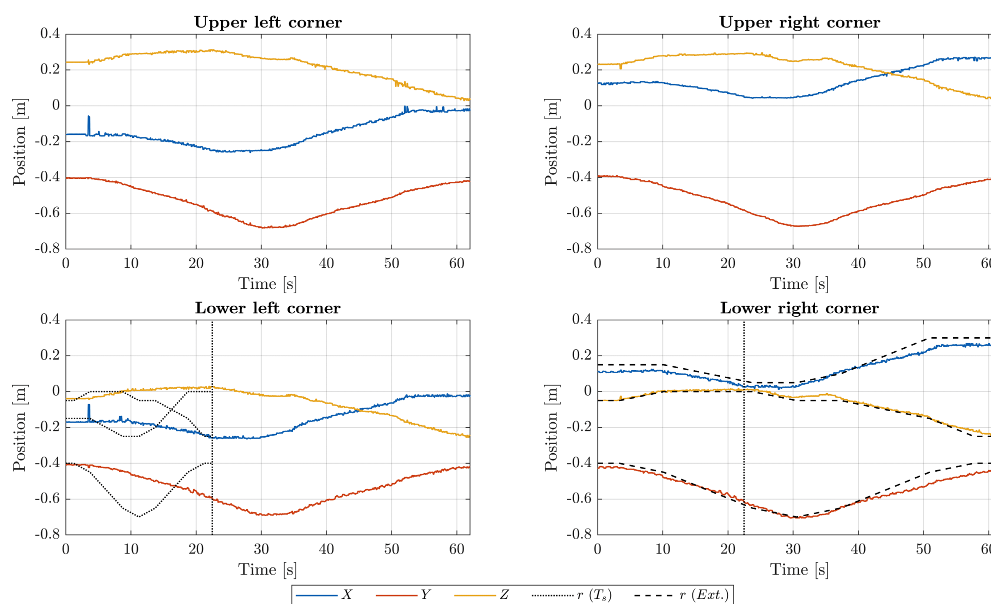

Figure 5.1: Experimental results without Vision feedback nor running in real time

These results were to be expected, as this was implemented with an early version of the MPC without some of the improvements and tuning explained in the previous chapters (specifically the final structure and weights detailed in Section 3.3), and of course each iteration had to wait for the optimizer to finish before proceeding. Additionally, all the debugging commands, timing processes and printing data through terminal, were enabled to check the correct functioning of the system, adding more time per iteration.

Besides the extended time, the evolution of the cloth and the tracking results were satisfactory even with just the MPC node working with optimizer and backup SOM, so the implementation continued adding feedback coming from the Vision and Processing nodes, as explained in Section 4.4. Figure 5.2 shows an example of the results obtained in this situation, with a full closed-loop scheme but still without running in real time.

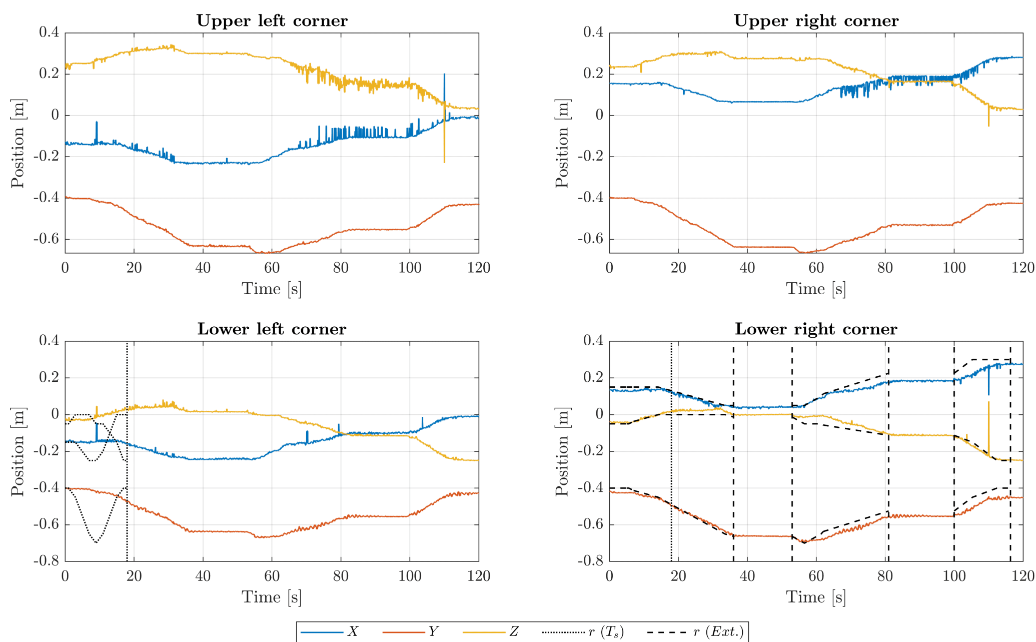

Figure 5.2: Experimental results with Vision feedback but not running in real time

In this case, and all the experiments carried out in this scenario, we observe another particular behavior. The shown example was done using $T_s = 0.020$ s, so the trajectory should theoretically end after 18 s, as represented in the lower left corner. However, not only does it take much longer (ending around 6.47 times later, at 116.5 s), but in some iterations, the optimizer took times in the order of seconds or tens of seconds to compute the next control signal, halting the execution completely during that time. This is indicated in the lower right corner plot, with two zones without any reference between vertical separators.

Even with stretched and paused executions, the tracking itself was still correct, with the overall shape following the reference trajectories when extended in time. These results lead to the changes explained in Section 4.5 to guarantee that, first, the optimizer did not block the overall execution, and second, if the optimizer got stuck due to noisy or demanding new initial states, it would timeout before it was too late and start over with new initial parameters.

In other words, it was empirically proven that explicit changes were needed to run in real time. Once those were in place, new experiments were performed to prove the new setup worked correctly. Fig. 5.3 shows an example of these experiments, where a longer and more demanding trajectory was tested (in fact, it can be seen how it is an extension of the previous one, as the first movements are the same as the full previous trajectory).

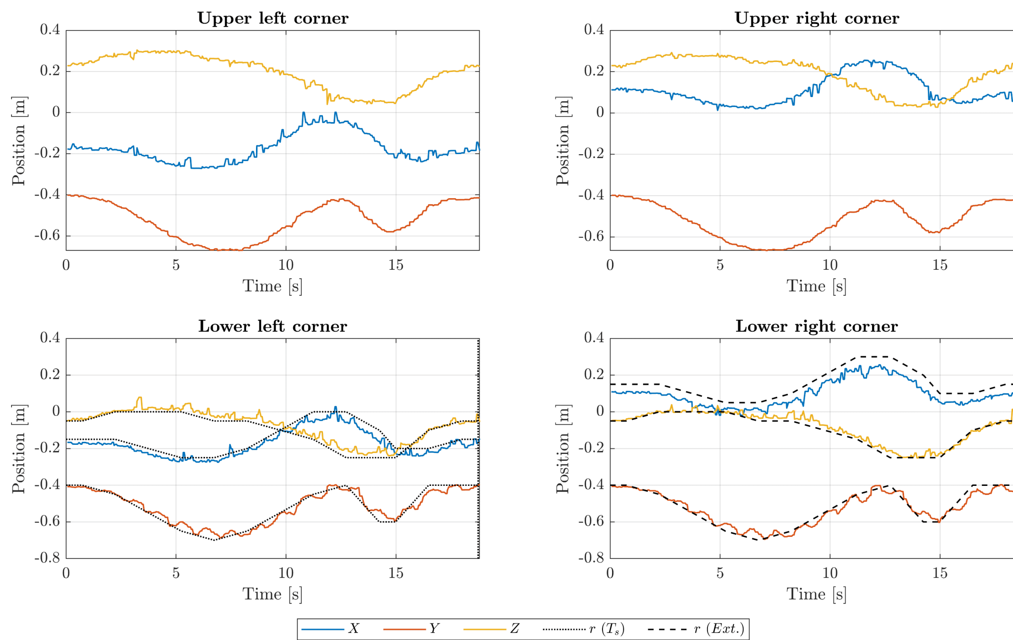

Figure 5.3: Experimental results with Vision feedback and running in real time

In the shown case, the time step was $T_s = 0.015$ s, to prove how even with faster rates than the ones shown before, the system could keep up and finish the execution exactly at the same time as the theoretical end of the reference trajectory, at 18.75 s. We have kept the two different references in the plots to show how the "extended" version is exactly the same as the one following a constant rate. Furthermore, the tracking performance is still unaltered, even with faster rates and trajectories.

In fact, checking the main KPI for tracking performance, the resulting RMSE, it is clear to see how once the Vision feedback was implemented, it slightly but consistently increased in all experiments (comparing against stretched and paused references), but with the new real time implementation, it improves again. For the shown experiments, without feedback it had 3.9 cm of RMSE, the second case with feedback but without real time went up to 6.7 cm, and the final version, even with a much more demanding situation, goes down to 5.5 cm. Even if other experiments reached the same levels of error as without adding feedback, this one is shown to demonstrate the improvements allow executing more adverse conditions.

In any case, we can see how the output data for these experiments is quite noisy, and does not exactly represent the evolution of the cloth, leading to larger errors. These experiments were all carried out without any kind of filtering on the feedback data, as the filter selection had to be done once the scheme ran in real time.

5.2 Vision Feedback Filter Selection

Sending the raw mesh data, with the only processing being an ordering and a change of base, directly to the MPC to update the initial state for the optimizer can lead to it getting stuck and not finding an optimal solution in time, and reaching a timeout. Even when the data is combined with the evolution of the backup SOM using the W_V weight to reduce this effect, and distances over a certain threshold are discarded immediately as spurious signals, after some iterations, it was observed that this effect still persisted in real experiments.

This is why a filter was required, even if it was at the cost of some time delay on the feedback signal. The considered alternatives are explained in Subsection 4.4.3. They are all based in the concept of Moving Average using previous data, except the "Future" version, FMA, which waits for the next sample and then applies weights to the previous and next points to get a filtered value for the central one.

After some experiments testing different alternatives, it was seen how considering only the previous one or two points had resulted in no significant improvement, while more than four increased the delay between capturing the image and the data reaching the MPC too much. The definition of the EMA is recursive, as seen in (4.4d), which results in a filtered output depending on all values from before in exponentially decreasing weights. To ensure data from too much into the past did not affect the current filtered values, the expression was truncated to the past three points too, resulting in the same filter as the WMA shown in (4.4c) with exponentially decreasing weights w_i. This is why only three filters were compared: FMA, SMA and EMA (truncated, same as WMA), and all of them considered three data points in total to compute the filtered output. Fig. 5.4 shows the evolution of the X coordinate of the lower right corner of the cloth executing the same trajectory with each one of the considered alternatives.

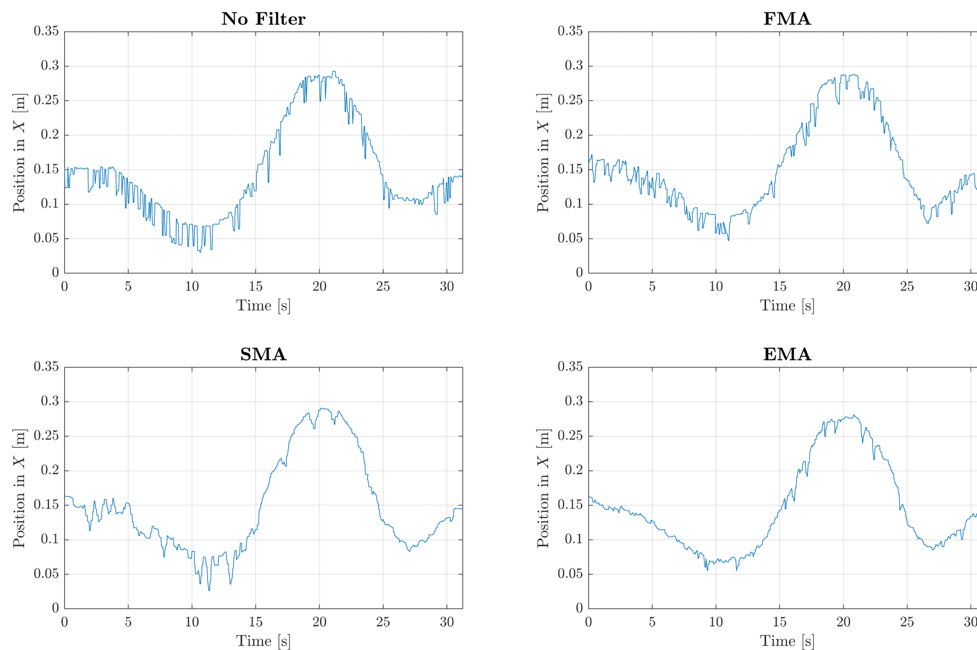

Figure 5.4: Evolution of the X-axis position of the lower right corner for all considered filters

First of all, these are different executions, done one after the other, changing the enabled filter in the processing node, and the EMA used $\alpha = 0.66$ (which, truncated at the three most recent samples, corresponds approximately to weights 0.66, 0.23 and 0.11). In the plots, focused on one coordinate to avoid cluttering the image, but representative of all of them, we can see how the unfiltered evolution is the most noisy, and how each filter reduces the variability, with SMA and EMA being the ones that produce the most noticeable changes, qualitatively, in the shown results.

In fact, the evolution in the X direction of this corner has been chosen specifically because, besides the regular noise present in all coordinates and nodes, we can notice jumps between the actual corner position and another value, always at approximately the same distance from the real position. Investigating the resource of this specific noise, it was found that even without the cloth moving, the Computer Vision algorithms used to detect the cloth and produce a complete mesh sometimes did not detect the lower corners precisely. This resulted in the corner node being placed close to its neighbor on the same row, producing an error mostly in X. This effect was more severe and common in the right lower corner than in the left one, producing jumps between the two situations shown in Fig. 5.5. This is also the reason behind the persistent offset between the obtained X of this corner and the reference seen in Fig. 5.3.

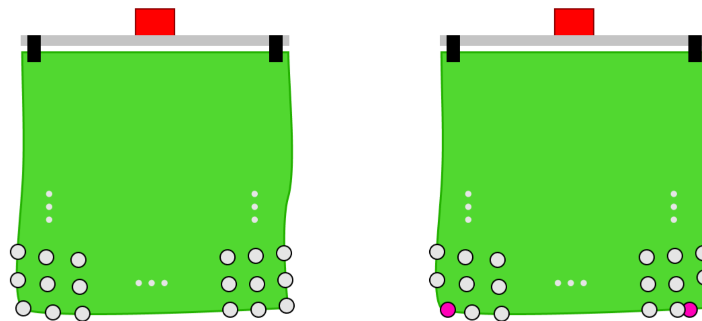

Figure 5.5: Detail of the found situation where the corners of the Vision mesh are not placed accurately

As this problem was originated in the Vision node, solving it on the root is out of the scope of this Thesis, but a way to work around it had to be found. Given the obtained data jumps between the correct value and an offset, and that the cloth is inextensible and the reference trajectories are placed at a constant distance equal to the side length (30 cm in the performed experiments), a good filter can reduce the effects of this phenomenon. Furthermore, if the output data captured by the camera and processed by the Vision node shows an offset between reference and results in only one coordinate of one corner, we know the cloth has a constant length, and thus this difference is either because the cloth is folded, not completely extended, hiding the real corner from the camera, or an error coming from the described source. In the performed experiments, where the cloth piece was held vertically and always extended, it could only be the latter. Even if this data is not completely filtered and reaches the optimizer, it will assume the corner is slightly folded and compute the control signal correctly, just with some added tracking error not present in the actual cloth. Finally, given the Vision node was developed at the IRI, this issue is now known, and the node was updated to also feature SMA and EMA filters before publishing the mesh (in a new topic, `cloth_mesh_filtered`), which was proven to produce less total delay experimentally.

Once this behavior was acknowledged, given we still had to work with it, it was assimilated as an added noise particular to this corner and coordinate that the filters had to deal with and reduce its effects. Comparing filters is still fair, as the input data presents this behavior in any case, and even if it adds some artificial error, it will be a constant baseline for all of them. To choose the final filter, a total of 4 experiments were executed for each one, plus also for the system with no filter. The conditions for each one of them can be seen in Table 5.1.

Table 5.1: Conditions of the filter selection experiments

Exp.	T_s **[ms]**	H_p	W_V
1	10	30	0.5
2	15	30	0.5
3	25	20	0.2
4	25	20	0.5

For the case without any filter, only the last two experiments reached the end of their executions, with the first two having many consecutive timeouts of the solver, and a sudden movement on the successful optimization that produced a safety stop of the Cartesian controller. This is why in Fig. 5.6, where we show the obtained RMSE for all experiments, there are only two dots on the NF column.

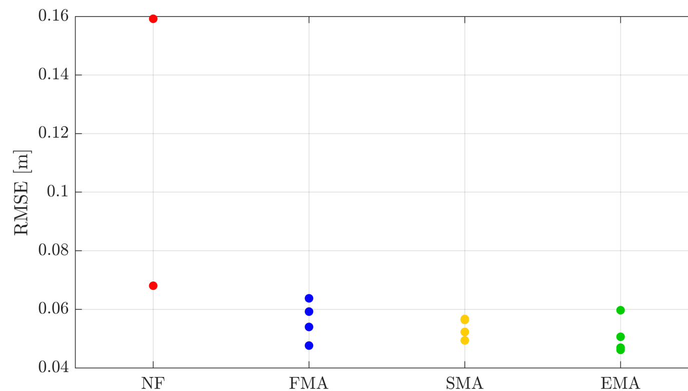

Figure 5.6: Results obtained on the filter selection experiments

Even with just the experiments that finished, the resulting errors without filtering are the highest ones, as expected. Between the three filters, FMA has the worst single result (with Exp. 4) and the highest variance in results. Waiting for the next sample and averaging the last three has approximately the same effect in terms of added delay or lag, as shown in Fig. 4.8 (in motion, the average is a better approximation of the previous point than the current one, making SMA like a shifted FMA. Having SMA on the Vision node helped add this offset in time), so with the results obtained, SMA is preferred to FMA. EMA got the best single result (with Exp. 1), and all of them are under 6 cm of RMSE, with a variability a bit larger than that of SMA. Between these two, EMA was finally the chosen filter and was used in all the following experiments.

5.3 Analysis of Control Parameters

With the system working in real time and the most suitable filter selected, a complete analysis of the effects of T_s, H_p and W_V was performed, to find the combination of these parameters that yielded the optimal tracking results. This study had to be done with experimental results obtained with the modifications only implemented in the real setup, as the developed simulations do not have an optimizer running in parallel with a strict timeout condition to ensure working in real time, and do not work with real feedback data captured online. The analyzed parameters precisely affect execution times and the reliability of the feedback data coming from the Vision node, which are different in simulation and in the real implementation.

Before that, however, there is still another variable that affects the behavior of the system: the mesh side size n for both linear models, COM and backup SOM. In Section 3.2, we obtained the necessary parameters for all considered time steps T_s for two different sizes, 4 and 7 (16 and 49 nodes in total, respectively). The sizes were chosen to be able to use the smaller one as a sub-mesh of the larger, as shown back in Fig. 2.20. In simulation, both model sizes worked similarly, with the most notable effect of increasing size being a rise in computational time.

In the real setup, all combinations between model sizes can be tested to check for a pattern, and use the one that yields the best results, if they are significantly different. Given we have two possible sizes and two linear models, there are a total of four possible combinations in theory, but, as explained previously, the idea behind separating COM and SOM is to have a simpler and faster model inside the optimizer without losing the main dynamics of the system, thus making the COM larger than the SOM goes against this idea and this combination was discarded. The other three were executed in 5 different situations, obtaining the results shown in Fig. 5.7.

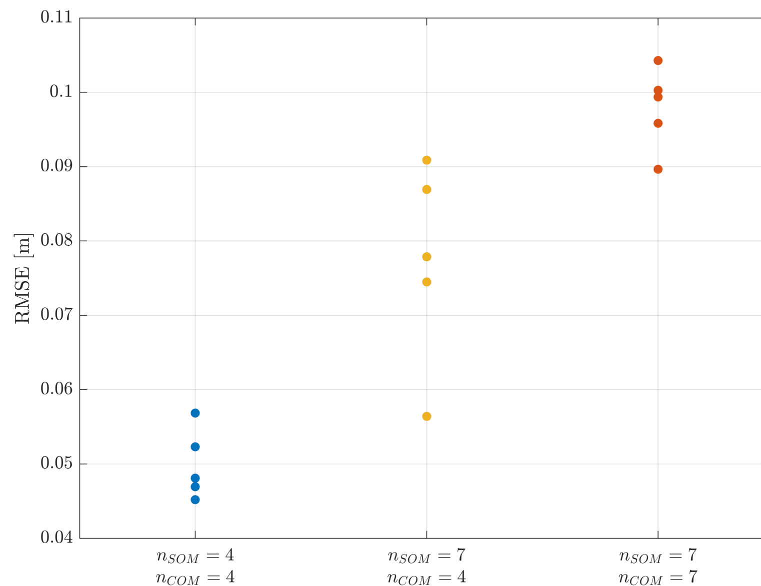

Figure 5.7: Obtained RMSE using different linear model sizes (5 experiments each)

It is clear how using the smaller models yields the optimal results. In fact, observing the executions of all experiments, this is actually due to the increase in complexity with $n = 7$, which produced a rise in computational times and several timeouts, accentuated even further when both models used the larger size. In fact, the experiment with the least error in both cases using $n = 7$ was the one using $T_s = 25$ ms, where the optimizer had more time to reach the optimal solution and output an updated control signal. For the full analysis of the three considered parameters, both linear models were chosen to have a size of $n = 4$, seeing how this produces the optimal results in general, for multiple combinations of them.

The range of possible values for T_s also comes from the obtained parameters in Section 3.2. Concretely, we can simulate at steps of 10, 15, 20 and 25 ms. The prediction horizon H_p can take any arbitrary (positive integer) value, but thanks to the experiments performed in simulation (for example, the ones shown in Fig. 2.24), we have an indicative range with relatively low errors without increasing computational times over the limit. As the conditions in the real setup are different with regards to timing and feedback data, in the analysis horizons were tested from 10 to 30 steps, in increases of 5 (10, 15, 20, 25 and 30). Finally, the weight W_V was tested in increases of 10% from 0 to 50%. While higher values were also tested, the majority of combinations resulted in unsuccessful executions. This results in a total of 120 finished experiments under the same conditions except these three parameters. After they were done, however, some additional changes were made to test executions with higher W_V. They only worked after reducing the Cartesian controller gains and using a filter with $\alpha = 0.5$ and 4 points, but all cases with $W_V = 1$ (100%) finished correctly. This makes a total of 140 experiments, with all their results shown in Fig. 5.8. Given the conditions for $W_V = 1$ had to be different from the rest, no values of W_V were tested in between to keep them separated and focus on the 120 executed with the same conditions.

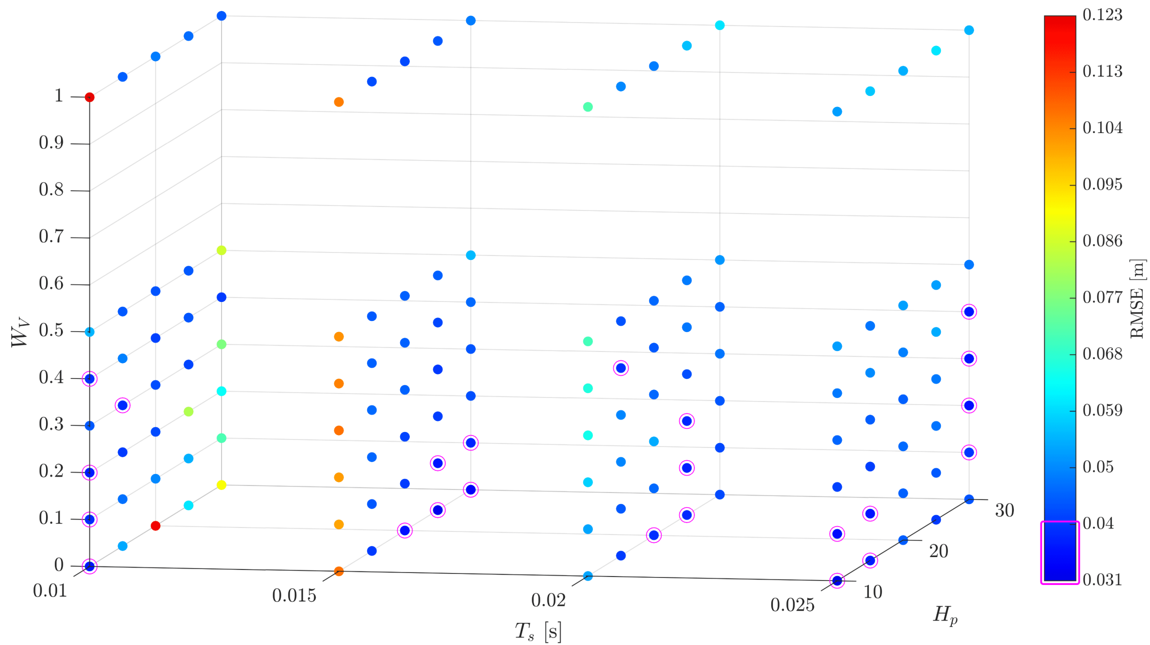

Figure 5.8: Results of all the executed experiments depending on T_s, H_p, W_V

In fact, even after adapting the conditions, the final experiments with $W_V = 1$ yielded worse results than their corresponding version with $W_V = 0.5$, almost in all cases, but the absolute worst result ($T_s = 10$ ms, $H_p = 10$, $W_V = 1$) is only 2 mm worse than the previous worst ($T_s = 10$ ms, $H_p = 20$, $W_V = 0$), that had 12.1 cm of RMSE. This is also a reason why we plotted all of them together, as the color scale was almost unaltered including the final 20. In the plot we have also marked with a magenta circle all results within the best 10% of errors, i.e., with an RMSE lower than 4 cm.

With these results, we can clearly see how a low T_s combined with a high H_p does not produce correct tracking. During execution, these experiments produced several timeouts on the optimizer, trying to keep up with a very fast rate while predicting a lot of steps into the future. Actually, some other issues were also seen during the execution of all experiments with $T_s = 10$ ms and $W_V \geq 0.3$, thanks to the debugging feedback printed on the command terminal. In these cases, instead of the optimizer timing out, the feedback data was consistently being discarded either for it being too much into the past (over the saved control signal history used to update it) or too distant to the simulated state of the backup SOM. This means that these experiments actually have an effect of the feedback data closer to $W_V = 0$ than their actual value, as most of the time the real data was unusable. Furthermore, a lower time step results in an increased difference between the rate at which the control signals are computed and sent to the robot and the rate at which the Vision node outputs feedback data, being around 10 times slower in these conditions. This means that the computed errors are also less reliable, as there are fewer captured points within the same trajectory to compare with the reference. With all this, even if we have some results with really good tracking errors using $T_s = 10$ ms, it is clear that this sampling time is too fast for the majority of cases, and must be avoided to obtain optimal tracking results.

For both $T_s = 15$ and 20 ms, we clearly see how a horizon of 10 steps is too short to track correctly, regardless of W_V. This effect disappears with $T_s = 25$ ms, having optimal results with the shortest H_p too, which means that it is a problem related to total prediction time and maximum allowed time for the optimizer. Even if $T_s = 10$ ms also has good results with $H_p = 10$, we have discussed how these results are not as reliable with the conditions of the executed experiments.

We can also see a tendency of errors increasing with higher W_V, with some optimal results being obtained without considering the Vision feedback at all. This is of course a product of the noise, present even after filtering. In the executed experiments, there were no strong rotations, sudden movements, offsets, nor other disturbances (e.g., wind or human actions), and the backup SOM always started in the exact same position as the real cloth, making its simulated evolution an accurate one without any added noise. Of course, $W_V = 0$ means the control loop is not actually closed, and cannot be applied in a general scenario, where external forces or initial deviations can make the SOM state have the unreliable evolution, and the real feedback would have to correct its state. Unfortunately, with the current camera and Computer Vision algorithms, this comes at the cost of updating the initial state of the MPC with noisy data, which can increase optimization times. A general application of this scheme would need a camera with a faster refresh rate, more precision, not fixed in place to enable more movements and orientations without losing the cloth, and a fast and reliable algorithm to obtain an updated mesh.

All in all, we can see how even discarding cases with $T_s = 10$ ms, $W_V = 0$ and 1, and the discussed cases with $H_p = 10$, there is no concrete singular region that yielded optimal tracking results (errors lower than 4 cm, the 10% selected before). However, all the remaining cases, 65 different combinations ranging from $T_s = 15$ to 25 ms, $W_V = 0.1$ to 0.5 and $H_p = 15$ to 30, plus $H_p = 10$ for $T_s = 25$ ms, yielded errors lower than 5.5 cm, which is not far from the previously considered threshold, and an acceptable error considering the large range of options it includes, and the precision of the camera and Vision algorithms used.

Choosing $T_s = 20$ ms, $H_p = 25$ and $W_V = 0.2$ as an example of a combination that yielded optimal tracking, with an RMSE of only 3.8 cm, we can plot the evolutions of all corners, as shown in Fig. 5.9. We can see how some noise persists, and there is a slight offset in X on the lower right corner, but the reference is tracked correctly. In this figure we also include the evolution of the TCP positions, comparing the Pose command $\mathcal{P}_{TCP}$ with the actual positions given by ROS (using FK).

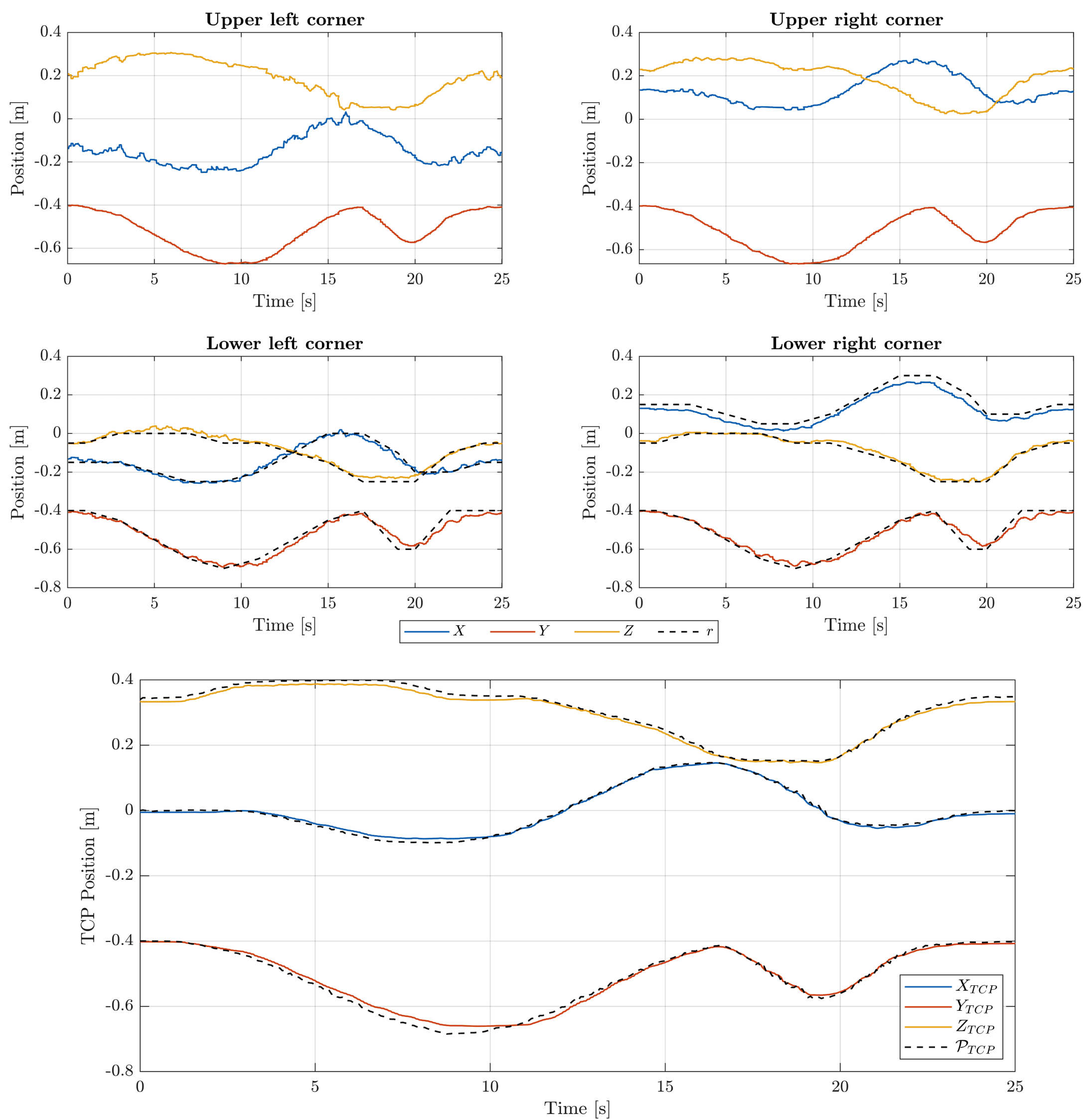

Figure 5.9: Cloth corners and TCP evolutions for $T_s = 20$ ms, $H_p = 25$, $W_V = 0.2$

5.4 Tracking in Adverse Conditions

This final section is dedicated to showing and analyzing results obtained by executing in conditions that can be more challenging for the optimizer, starting with trajectories with changes of orientation. Next, the effects of executing the same trajectory at different speeds are analyzed. Finally, we show the results obtained in experiments with heavy disturbances, such as blocking the camera or applying forces to the robot arm manually.

First, in Fig. 5.10 we show the evolutions of all cloth corners and TCP for a trajectory similar to the ones shown previously, but instead of returning to the starting point, it ends with a rotation of 45 degrees followed by a counter rotation of 90 degrees. In the TCP evolution, we not only show the positions, but also the orientations expressed as a quaternion, $[\epsilon_X, \epsilon_Y, \epsilon_Z, \eta]$.

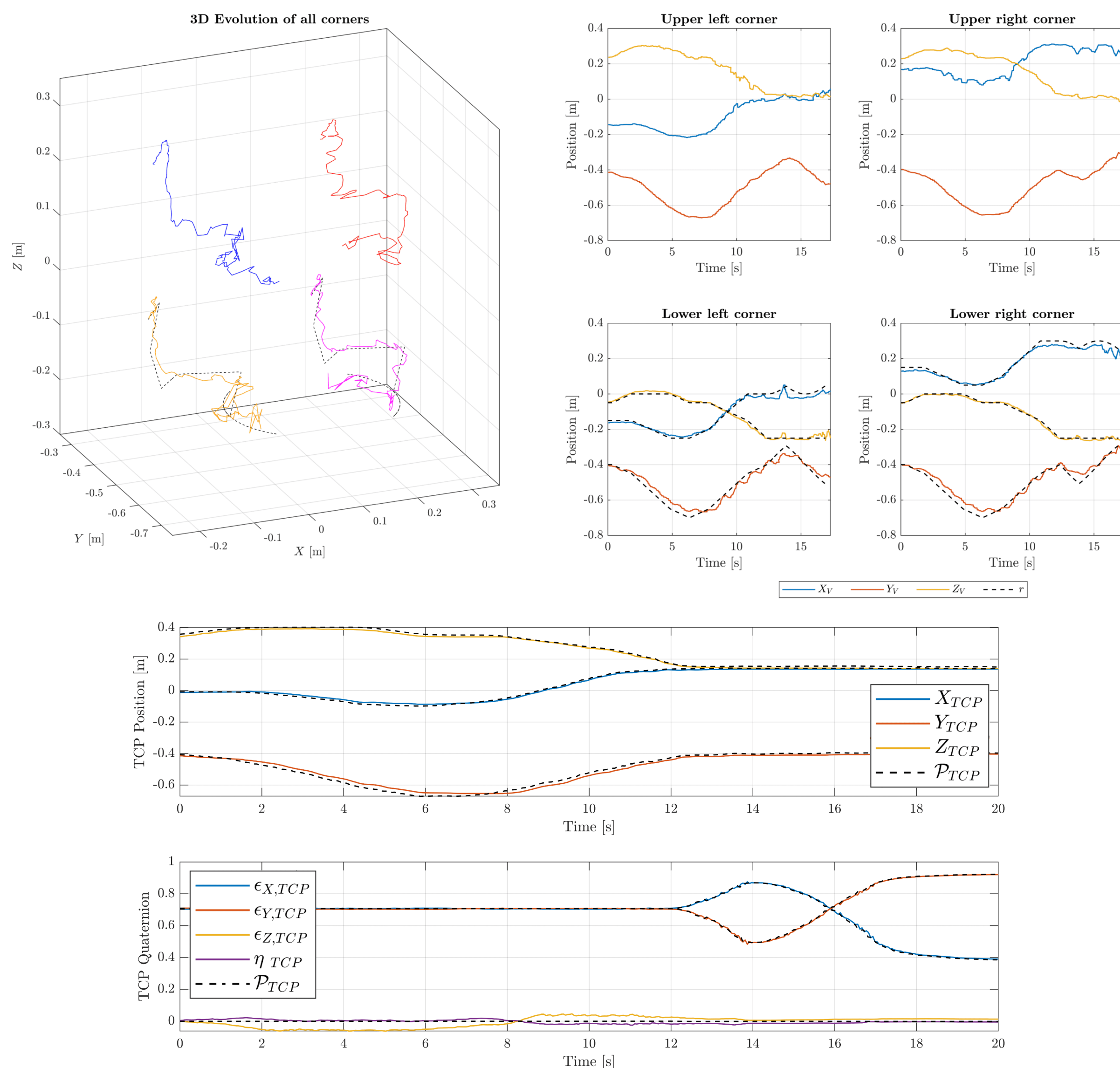

Figure 5.10: Cloth corners and TCP evolutions tracking a trajectory with rotations

This is just an example with a rotation around the Z_E axis of the TCP (which matches the global Z-axis, in this case), but similar ones were executed around the axis perpendicular to the cloth plane with similar results. It is clear how the trajectory is tracked correctly, with the expected noise coming from the real feedback data, especially visible in the 3D plot. In this plot we can also see how some of the harder corners with immediate direction changes affecting all coordinates are also cut more smoothly in the resulting evolution. Having a translation first and a rotation after, we can clearly see how the TCP evolution keeps a constant orientation while moving and then the same position while rotating back and forth. Once again, in the TCP evolutions, we show the pose command sent to the Cartesian controller, $\mathcal{P}_{TCP}$ (with both position and orientation) and the actual evolution as given by the messages published on the ROS topic `iri_wam_controller/libbarrett_link_tcp`. In summary, we can see how the robot tracks trajectories with rotations correctly, even those that set the cloth in positions challenging for the camera to see.

Next, we can analyze the effects of changing the time step with regards to speed. The chosen method of execution was to have trajectories without time stamps on them, and simply use the next point on each iteration. This results in the same trajectory being faster or slower depending on T_s. To compare the effects of changing T_s while moving at the same speed, some new trajectories were made, exactly as previous ones, but keeping only every other point, making them twice as fast in practice.

As an example, Fig. 5.11 shows a comparison between the original trajectory executed at T_s = 10 and 20 ms, and the new fast trajectory at 20 ms. For the latter, two values of H_p were tested, as increasing T_s and keeping H_p the same increases the total prediction time. We can see how the original slower trajectory lasts 12.5 s at 10 ms per step, while it takes double this amount, 25 s, at double the period. Meanwhile, the fast trajectory takes 12.5 s when executed at 20 ms per step.

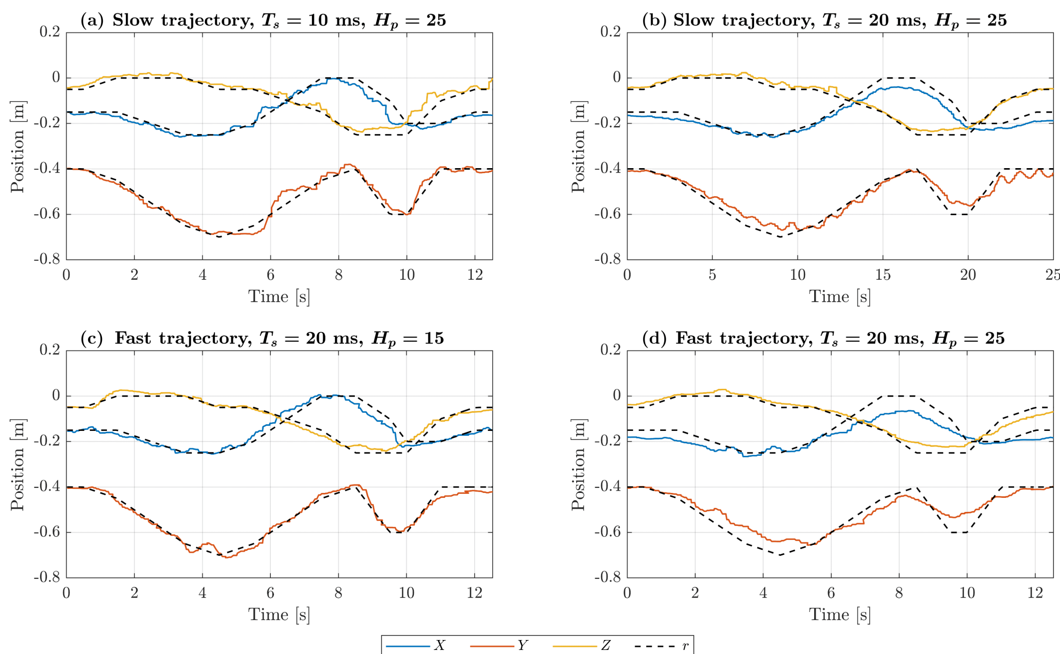

Figure 5.11: Comparison between evolutions of the same cloth corner executing at different speeds

Executing at $T_s = 10$ ms (Fig. 5.11a), we get several timeouts around 4 s, which results in the robot not receiving updated commands and not moving, followed by a fast movement around the 6 s mark. The timeouts happen because the solver cannot find the optimal solution in the short amount of allowed time. This is solved using $T_s = 20$ ms (Fig. 5.11b), but with the same amount of steps, we now have a trajectory that takes twice as long to complete, meaning not only the solver had more time to finish, but movements were slower, making it easier for the Vision node to capture, process and publish the data. Between these two cases, even with different total prediction times (0.25 vs 0.5 s), the total displacement from the initial state of the prediction until the end of the horizon is kept constant, and so are the displacements per step, as we have the same displacement in the same steps, the only difference is that the steps are longer.

If we keep $T_s = 20$ ms and $H_p = 25$ but change to the trajectory using only half the samples, the obtain the plot on the lower right corner (Fig. 5.11d), where we can see that the Y coordinate is behind the reference (a fast movement makes the lower corners “lag” behind the top ones) until it changes direction. Additionally, we can see how some back and forth motions are cut short both in X and Y near the end. In this case, each optimization problem has the same number of steps into the future as before, but we actually get two times the displacements from before, as half of the reference steps are gone. In other words, this execution has the same speed as the original one, moving along the same trajectory in the same time, but each optimization can see more into the future.

To have a completely fair comparison, we can reduce the number of steps in the horizon to match the original total prediction time. That would be cutting H_p in half, but that results in 12.5. We know that very low horizons, in the order of $H_p = 10$, were too short and resulted in larger errors, so we can use $H_p = 15$ for a total prediction time of 0.3 s instead of the original 0.25 s. This is the lower left plot, Fig. 5.11c, where we can see a better tracking, without the original timeouts, slowing the trajectory down, or cutting corners due to seeing too many steps ahead. In fact, this yields the minimum tracking error, at 3.6 cm, compared to the original 4.6 cm.

While having trajectories based on points and simply picking the next sample (sliding the window one sample) every step is a valid approach, keeps the same displacements per step, and makes higher T_s more favorable in the real setup, as they make the Vision node relatively faster (instead of 10 times slower, it becomes 5 times slower when going from $T_s = 10$ to 20 ms), it also makes the duration of the trajectory depend on the time between samples, which is usually not desired in real applications, where each position has an associated time to reach it, set from the start. The developed codes assume the references have no time stamps, but for future applications, this can be changed to consider them. For example, a reference parser could keep track of the current time since starting the trajectory and skip samples with stamps prior to the current time, giving the MPC the next sample to consider, keeping its side unaltered. The base for a Reference node was actually created following this idea, but its development was discontinued not long after, as the majority of experiments had already been done with the previous approach, and the effects of speed could be tested with new trajectories created by simply skipping samples of the old ones regularly, as in the shown example. Finally, once the used T_s is known, even trajectories with time stamps can be adapted to the used approach easily, so there was no urgent need of changing it.

The final part of this section is dedicated to experiments executed with disturbances and other situations coming from external agents. Concretely, two different cases were studied: blocking the camera, either by walking between it and the WAM robot or by covering it with a solid object, and applying forces to the robot arm itself during execution.

The results shown in Fig. 5.12 correspond to the first case, where, during execution, a human walked between the camera and the cloth, covering its view for about two seconds, and then walked back in the opposite way after a while covering the cloth again for approximately another 2 s.

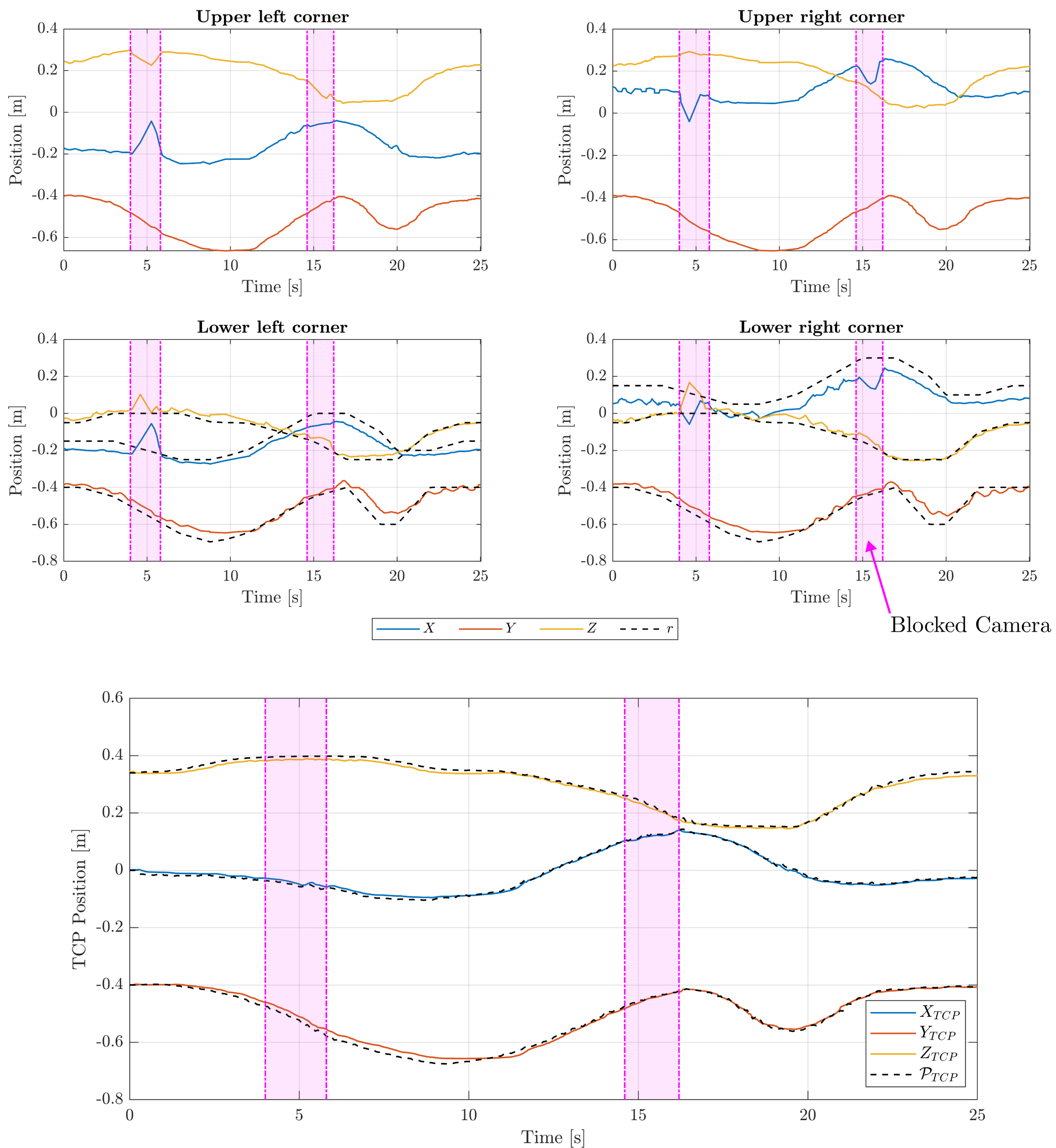

Figure 5.12: Cloth corners and TCP evolutions blocking the camera in two instants

The shaded areas correspond to the instants where the camera was blocked, as indicated in the figure. These times have been obtained thanks to a video recording of the experiment, but can also be double checked with the recorded ROS messages published by the Vision node, which either stop or have strange behaviors. When covering the cloth only partially, the Vision node can still detect its visible part, but the resulting mesh can have incorrect positions on all nodes and coordinates. Usually, depth (global Y) is the most reliable output in these situations, as it sometimes tries to assign an entire mesh to just the visible part, moving the other two coordinates around. This can be clearly seen in the plots, where on the first blocking, around the 5 second mark, a new mesh was published after about a second with none, but it had completely incorrect X and Z values for all corners. A similar behavior can be observed on the second block, past the 15 second mark, but is only pronounced in the right corners.

This sudden change with a high distance is caught by the feedback processing done before updating the state of the linear models, and discarded completely, so even if we see it happen in the raw and even filtered feedback data, it does not affect the controller. A proof of this is that we can see how the evolutions of the TCP are completely smooth even during these moments. Of course, this can only be done thanks to the backup SOM inside the controller, which accurately simulates the real evolution of the cloth during the moments where the Vision feedback cannot provide updated data. Before continuing, Fig. 5.13 shows a picture of one of this situations, extracted from the aforementioned video of the experiment.

Figure 5.13: Human walking in the setup, creating a barrier between camera and cloth

The second experiment with this kind of disturbances coming from external agents also included an instance of a person walking and momentarily blocking the camera and Vision feedback for a couple of seconds, but it also involved more direct interactions with the WAM. As mentioned in Section 4.3, the used Cartesian controller allows movements in the null space of the WAM without offering much resistance. This space comes from the fact that there are multiple joint configurations that result in the same End Effector pose, meaning the IK problem does not have a unique solution, and thus some joint motions exist such that, combined, do not alter $\mathcal{P}_{TCP}$. This has multiple applications, but for the current setup and tracking problem, we can also use it as a source of disturbances.

For example, during execution, a human can pull and push the elbow joint like shown in Fig. 5.14 with almost no effort, and while theoretically these movements do not change the TCP pose, as the Cartesian controller keeps it in place and only allows a movement in the null space, during a real experiment, the are slight displacements and forces that are transmitted from the arm into the cloth, producing disturbances that are picked up by the camera.

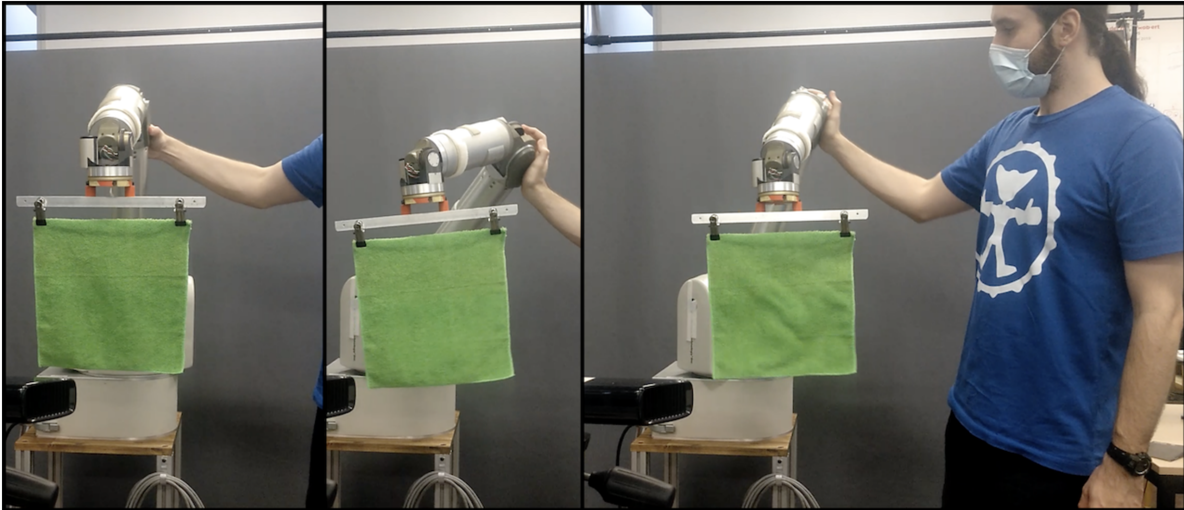

Figure 5.14: Human agent pulling and pushing the elbow joint during an execution

In fact, the second captured experiment not only had this kind of movements in the null space causing slight disturbances, but after some seconds, the rigid piece connecting the upper corners of the cloth was pressed down on one end, causing a slight rotation and a vibration when released. This situation can be seen in Fig. 5.15. These last two pictures are frames extracted from the same video recording.

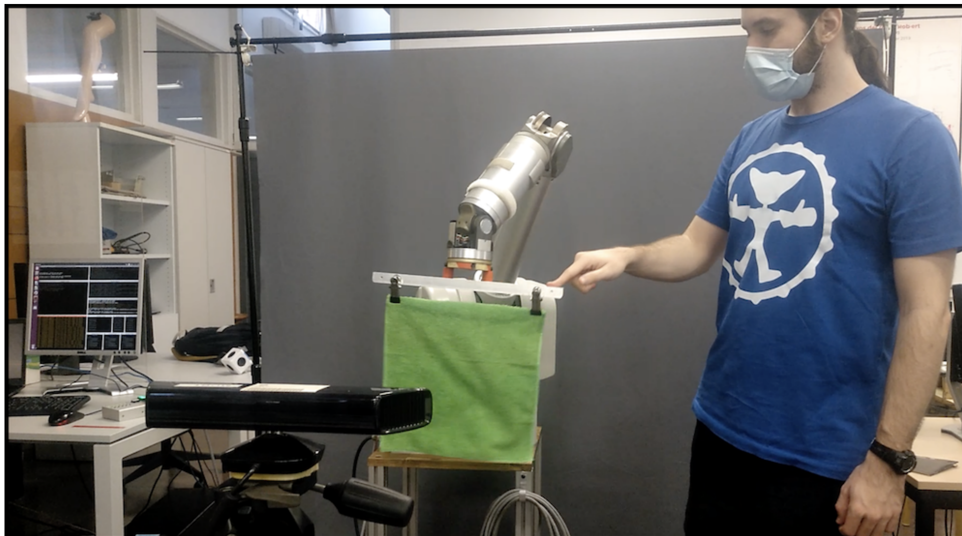

Figure 5.15: Disturbance created by a person poking the cloth

With the conditions of this second experiment explained, we can now show its results, which are in Fig. 5.16. The shaded region in magenta corresponds to the interval when the camera was blocked, as in the previous experiment, while the region shaded in gray corresponds to the time when a human agent was interacting with the robot arm.

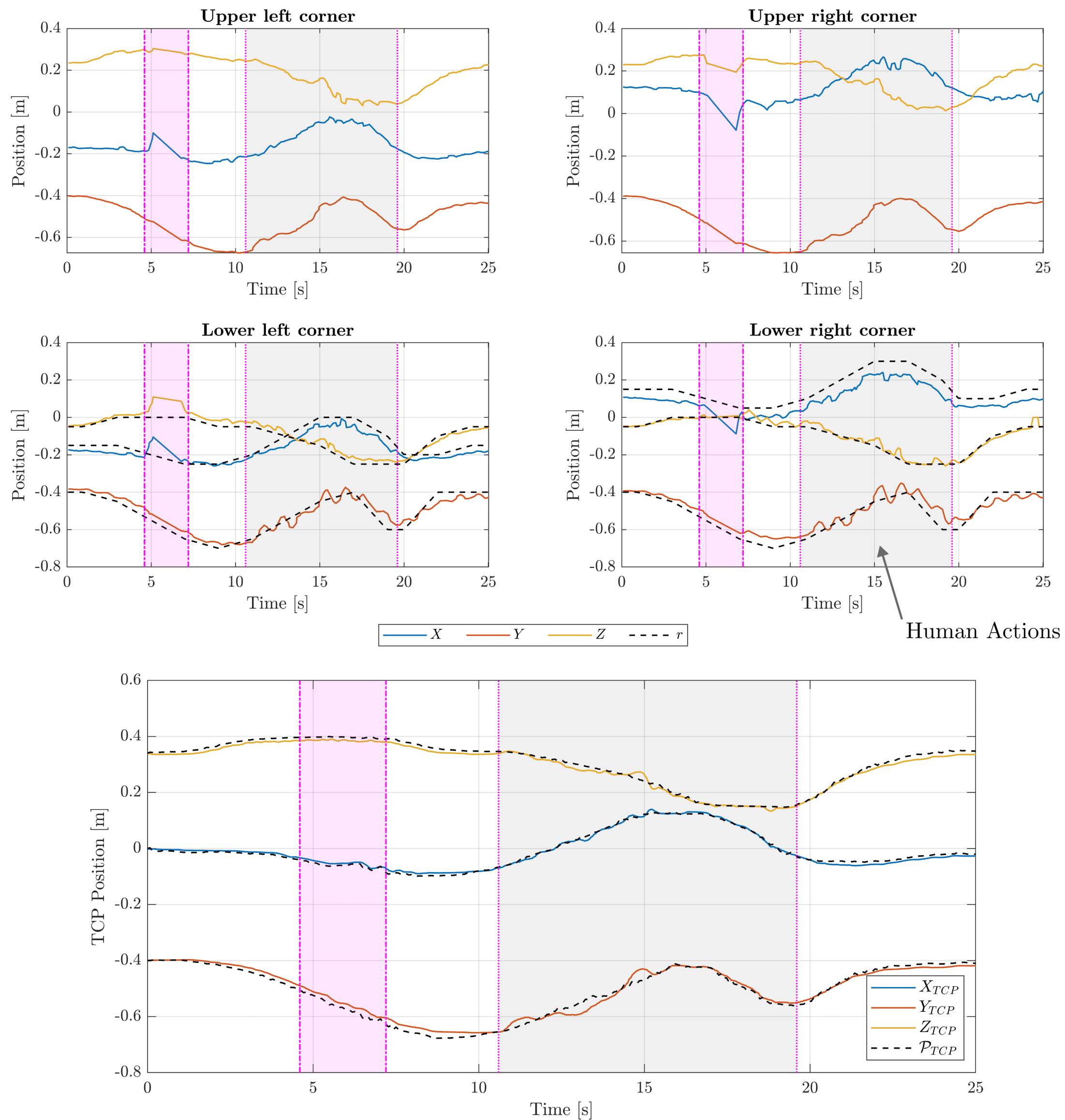

Figure 5.16: Cloth corners and TCP evolutions blocking the camera and with human interaction

Blocking the camera has the same effects as before, but now we see different results with the new actions. Moving the elbow of the arm (from around 11 to 15 seconds) produces slight movements in the TCP, which are also visible in the upper corners, and are propagated to have a much greater effect in the lower corners due to the non-rigid nature of the cloth. When the rigid piece between the upper corners is pressed and released around the 15 second mark, we can see how it creates a ripple effect which is very clear in the upper corners, and somewhat mitigated on the lower ones. Even in these situations, however, we can see how the reference trajectory is tracked correctly.

All in all, the developed final implementation on the real robot has been proven to work with demanding conditions, in multiple situations, and even under the effects of an external agent creating disturbances while keeping a good trajectory tracking performance, with these last two experiments having RMSEs of 7.1 and 8.0 cm, respectively.

6. Budget and Impact

This chapter focuses on the resources used during the course of this Thesis and its effects in society, both in the economical sense, with the budget discussed in Section 6.1, and the ecological sense, with the environmental impact detailed in Section 6.2.

6.1 Project Budget

The total budget can be divided in material costs, including hardware, its amortization and software licenses on one side, and personal costs on the other.

For the latter, we first have to consider that the present Thesis had a total duration of 7 months, from mid-February to September 2021. It is important to note that while this Thesis was awarded with a *"María de Maeztu Unit of Excellence"* Research Initiation grant, this only covered four of these months, from March to June, with an endowment of 525 € per month, assuming 20 hours of work per week. However, while this is the real funding obtained, it does not cover the total costs of the project, and not even only the personal costs if we compute the budget as if this Thesis was an independent, private project.

The present Thesis corresponds to the final work of a Double Master's Degree, and therefore has an assigned total of 24 ECTS (European Credit Transfer and Accumulation System) credits. At the UPC, each credit corresponds to 25 to 30 hours of work. While the usual for regular subjects is 25, in Thesis and internships it can be increased to 30 h/credit [56], thus this value will be used for the budget computations, resulting in a total of 720 hours.

In addition to the hours dedicated by the student, which will be assumed as a junior engineer with a wage around 20 €/h, it has been estimated that the principal researchers devoted 100 hours to the project during the course of this Thesis, with a unit cost of 38.21 €/h.

With regards to material costs, there were no purchases of new equipment needed, as all hardware was already present at the IRI, except the additional personal computer used, which was also not bought specifically for this Thesis. However, the usage of this material carries a depreciation, which must be accounted for in the total budget. The details of amortization periods and values were facilitated by the organization office at the IRI itself.

We can start with the WAM robot, which was bought for 120000 € and has an estimated amortization period of 20 years, corresponding to around 500 €/month. However, this robot also needs some periodic maintenance, and historic data show that these tasks plus the regular changes of wires and other parts represent an additional cost around 18 €/month.

The robot is connected to a computer at the laboratory, bought for 600 €, which for an estimated lifespan of 10 years, corresponds to 5 €/month. Additionally, for the experiments using the Vision ROS node, an additional computer was used with a better graphics card, with an estimated depreciation of 6 €/month.

For these experiments, the camera used was a Kinect (Xbox 360 model), bought originally for 150 €. It is expected to last for 5 years, which supposes 2.5 €/month.

Finally, a personal HP laptop was used, with Windows 10 as the operating system. It had a price of 1700 €, which for a maximum lifespan of 10 years, means 14.16 €/month of depreciation.

Besides the used hardware, the software used to develop this Thesis included Ubuntu 16.04, ROS Kinetic, the CasADi and Eigen libraries, and Latex (Overleaf), all open source and free. For some figures, Figma was used in its free of charge Starter plan. Other additional codes were developed at the IRI, as mentioned and cited through this document, also without any monetary cost. The only licensed software used was Matlab, and its Academic License, which can be used for research projects, costs 250 €/year.

Table 6.1 shows the full budget of this final Thesis. For the amortization times, we have considered the actual periods in which each piece of hardware was used. For example, the robot and its PC were used during 5 months (March-July 2021, both included), but the experiments using the camera and requiring an additional PC were all performed in the span of only 3 months (May-July 2021). The final total cost of the project is 21106.50 €.

Table 6.1: Complete project budget

Concept		**Units**	**Unit cost**	**Total cost**
Personnel				
	Student	720 h	20.00 €/h	14400.00 €
	Principal researchers	100 h	38.21 €/h	3821.00 €
Amortization				
	WAM	5 months	500.00 €/month	2500.00 €
	WAM Maintenance	5 months	18.00 €/month	90.00 €
	WAM PC	5 months	5.00 €/month	25.00 €
	Vision PC	3 months	6.00 €/month	18.00 €
	Kinect Camera	3 months	2.50 €/month	7.50 €
	Personal Laptop	7 months	14.16 €/month	99.17 €
Software				
	Matlab Academic License	7 months	20.83 €/month	145.83 €
			TOTAL	21106.50 €

6.2 Environmental Impact

Developing a project always has impacts on the environment, either positive or negative, which must be studied in detail. Due to the nature of the research project developed during this Thesis, however, the total ecological impact is minimal, as no directly pollutant activity was carried out.

The used WAM robot, even if not purchased specifically for this Thesis, was built with Aluminum and PVC, which are both recyclable once the robot ends its useful life. The only negative impacts on the environment are derived from the energy of these recycling processes. This robot, and all the hardware used for this project, complies with the Restriction of Hazardous Substances (RoHS) regulation of the European Union [57], which forbids usage of certain elements, and especially Lead, in electric and electronic devices.

A negative impact on the environment can be derived from the consumption of electrical energy and the CO_2 emissions necessary to obtain it. We can assume that at least one computer was turned on during the time dedicated to the project, which is a total of 720 hours. The power rating of the personal laptop is 120 W [58]. With this value, we obtain a total consumption of 86.4 kWh during the course of this Thesis. Additionally, the WAM operates at least at 60 W [59], and was used around 400 h (5 months, but only 20 h/week), which results in 24 kWh consumed. The total electrical energy consumption rises to a total of 110.4 kWh. According to the official data published by the Spanish Government [60], a total of 0.357 kg of CO_2 are emitted to the atmosphere for each consumed kWh. With this factor, we can obtain that the total CO_2 emissions caused by the energy used during this project are of 39.41 kg CO_2.

The specific positive impacts of this Thesis are clearly defined with the possible applications of the developed controller, mainly in assistive robotics, as it has been developed under the CLOTHILDE project at the IRI [1]. For example, this control scheme could be used to aid a person with reduced mobility in daily tasks like getting dressed, folding clothes and putting them in the closet, prepare tablecloths, etc. While this can be a considerable impact in society and the lives of multiple people, the positive effects on the environment from an ecological point of view are more limited, and would depend on the energy and emissions saved using the developed scheme instead of another alternative on each specific application.

Conclusions

In this Thesis, a new Model Predictive Controller has been developed to be used in robotic cloth manipulation applications where a non-rigid cloth piece must track a defined reference without being fully grabbed by the robot. Reinforcement Learning techniques have been applied both to the cloth model and to the controller parameters to improve them finding the values that result in the optimal tracking performance. After being tested in simulation, the full control scheme was implemented in a real setup, where several experiments were executed to prove its tracking performance.

The main conclusions are structured around the defined objectives in Section 1.1, to compare the obtained results with the initial aims of the project.

The first objective that motivated this Thesis was to improve and generalize a previously developed MPC, tested only in simulation. This has been achieved by redesigning the controller completely, keeping only the cloth models themselves, and adding additional improvements progressively. We summarize the properties of the new controller in the following list, taking into account only the ones obtained before applying RL or implementing the controller in the real setup:

- The computational time for the optimizations in simulation was lowered from almost 50 ms to under 34 ms per step on the same conditions, and then to under 16 ms after tuning.
- The average tracking errors, measured with RMSE, were improved from around 6.7 mm to under 5 mm under the same conditions in simulation.
- The controller can now work with cloth models of any mesh size, provided that the parameters of the linear model are known. If multiple models are used with different sizes, the smaller one must be obtainable as a sub-mesh of the larger.
- Trajectories with rotations can now be tracked correctly. A simple and fast change of base has been proven to work well with the tested trajectories, but an alternative method is also provided in case more precision is required.
- It is particularized for the case where a single robot moves both upper corners of the cloth simultaneously, with an added quadratic constraint to the optimization problem to keep the upper corners at a constant distance, and computing a unique TCP pose between both corners. This can be easily removed or modified to match the necessary conditions, even in a situation where two robot arms are used.
- The slew rates (Δu) can be minimized in the objective function of the MPC instead of the pure control signals (u). They penalize sudden changes of velocity, which are also the moments where the linear model differs the most from the real cloth.

Another objective set at the start of this Thesis was to use Reinforcement Learning techniques to improve the behavior of the controller. This has been accomplished in two fronts: by learning parameters of the linear model for multiple time steps and sizes, to allow simulations and real executions in many combinations instead of just the original one at $T_s = 10$ ms and $n = 4$, and by obtaining the structure and tuning of the controller that yielded the optimal tracking results. This consists of:

- Minimizing Δu in the objective function, as it was proven that it reduced tracking errors.
- Using a constant weighting matrix Q_k in the objective function, without adding an adaptive term penalizing larger errors even further.
- Penalizing all coordinates with the same weight (having matrices Q and R as scalars by the identity matrix of size 6).
- A combination of weights with $Q = 1$ penalizing the tracking errors with the maximum value, and $R = 0.2$, to obtain optimal tracking in all the tested conditions safely, without risking using a value of R too low that could unstabilize the system.

The Relative Entropy Policy Search (REPS) algorithm was used successfully in both fronts, and a greedier version was developed to make learning processes faster in the specific case of tuning the controller once there was only one parameter to learn with a single optimum. The obtained tuning was checked under multiple conditions in simulation (T_s, H_p and even adding noise), with successful results and low tracking errors in the order of millimeters.

Finally, during this Thesis the last objective was to implement all the work done in simulation into a real robot, with all the necessary changes, to obtain a full closed-loop control scheme in a real setup. This has also been successfully achieved, adapting all the codes to C++ and later ROS, to connect all the needed nodes together, including MPC, a Cartesian controller, and Vision feedback and processing. Working with a real environment, hardware and data is much more challenging than executing simulations, and unexpected issues like the high amounts of noise in the captured images or the rate at which the Vision node processed them had to be addressed as they appeared. At the end of this Thesis, we have obtained a fully working control scheme in a real setup, which runs at a constant rate fixed by the chosen T_s, filters the output noise and has fail-saves against optimizations taking too long (timing them out) and vision data being too unreliable (with a backup SOM serving as a reference).

In the end, we can see how each one of these three objectives corresponds to the work described in Chapters 2, 3 and 4, respectively, and how all of them have been achieved successfully, finally converging in the experimental results presented in Chapter 5, where it is proven that the developed control scheme works correctly in all the tested situations, including less favorable ones like heavy rotations where the cloth is not clearly seen by the camera, faster movements where we are limited by the rate of the Vision node, and even situations where the camera was blocked for some seconds and the robot was moved due to external forces. With all these tests, we can confidently say that the Thesis has accomplished all of its objectives and the work done has been proven its performance.

There are, however, many more situations and options left to be explored. As a final note, we can list some of these possible future lines of research:

- **Find a general expression for the parameters of the linear model**. In this Thesis, we have seen how these parameters depend on both T_s and mesh size n, and found the values for some combinations through learning, but an interesting step could be to try to find relations between these two conditions and the resulting parameters, and with enough time and data, even a model that can be used for any combination within a set range of values.
- **Apply Online Learning**. The found tuning and controller structure has been proven to be the optimal one for the tested cases, but for a more general application, a learning algorithm could use recent data during executions to adapt the parameters of the controller to their optimal values specifically for the task at hand.
- **Test the Cartesian controller with Variable Impedance Control**. The original version of the used Cartesian controller, which was not a ROS node, included options to be executed with Variable Impedance Control (VIC). These options were lost when wrapping the controller as a ROS node to be able to use it as soon as possible, but a future research project could study the combination of MPC and VIC.
- **Use trajectories with time stamps**. For applications closer to the final intended purpose of this implementation, where the robot would interact with a human, or have to complete tasks in a set amount of time, instead of having references with a fixed number of points and a rate of points per second (the inverse of T_s), it would be more natural to have trajectories with key points to reach and time stamps indicating when to reach them.
- **Change the Vision setup**. This includes a faster and more precise camera, and a faster identification algorithm to obtain feedback data, but more importantly, changing from a situation where a single camera is fixed in place during the entire trajectory to another setup with either multiple cameras or mobile ones.

Acknowledgements

This Thesis would not be complete without mentioning and thanking everyone who supported me during its course, both academically and personally.

First of all, I would like to thank Dr. Adrià Colomé and Dr. Carlos Ocampo, for accepting me in this project, and letting me join the Perception and Manipulation Group at the Robotics Institute and the world of Robotics research. Their support, guidance and feedback have been invaluable and have made this Thesis possible.

From the Perception and Manipulation Group, I would also like to acknowledge everyone that helped me with insights, comments, or even previous research and codes. The final results could not have been obtained without the contributions of David, Franco, Sergi and especially Miguel, who helped set up the real experiments and was always ready to help with modifications to the Vision node.

Thanks also to my friends coursing the *Dobule* Master's at ETSEIB, for making this experience much more worth it, sharing pains and joys and helping each other, even more during the trying times of the COVID-19 pandemic.

I would also like to thank my lifelong friends Alberto, Marc, Aleix, Quel and Jaume, for their care and support through all these years, their interest and good wishes for this project. This is to all the times I had to miss meeting and enjoying time with you.

To Carlo and Bence, thank you for proving that even in distance, you can forge friendships and find people who share your passions and care about you. Thanks for your conversations and putting up with my random comments about this project.

Last but absolutely not least, I would like to thank my beloved family, for their interest and unconditional support not only during this Thesis, but all my life.

To all of you: Gràcies. Gracias. Thank you.

ETSEIB

Bibliography

[1] I. de Robòtica i Informàtica Industrial, "CLOTHILDE: Cloth manipulation learning from demonstration." https://clothilde.iri.upc.edu/, 2018. [Online; accessed July 2021].

[2] I. de Robòtica i Informàtica Industrial, "MP-CloL: Learning Robotic Cloth Manipulation based on Physics Models and Model Predictive Control." https://www.iri.upc.edu/project/show/245, 2020. [Online; accessed July 2021].

[3] J. Rawlings, D. Mayne, and M. Diehl, *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing, 2017.

[4] A. Colomé and C. Torras, "Dimensionality reduction for dynamic movement primitives and application to bimanual manipulation of clothes," *IEEE Transactions on Robotics*, vol. 34, no. 3, pp. 602–615, 2018.

[5] C. E. Garcia, D. M. Prett, and M. Morari, "Model Predictive Control: Theory and practice—A survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989.

[6] E. F. Camacho and C. B. Alba, *Model predictive control*. Springer science & business media, 2013.

[7] P. Tøndel, T. A. Johansen, and A. Bemporad, "An algorithm for multi-parametric quadratic programming and explicit mpc solutions," *Automatica*, vol. 39, no. 3, pp. 489–497, 2003.

[8] J. M. Maciejowski, *Predictive control with constraints*. Pearson education, 2002.

[9] R. Findeisen, L. Imsland, F. Allgower, and B. A. Foss, "State and output feedback nonlinear model predictive control: An overview," *European journal of control*, vol. 9, no. 2-3, pp. 190–206, 2003.

[10] G. Goodwin, M. M. Seron, and J. A. De Doná, *Constrained control and estimation: an optimisation approach*. Springer Science & Business Media, 2006.

[11] D. A. Allan and J. B. Rawlings, "Moving horizon estimation," in *Handbook of Model Predictive Control*, pp. 99–124, Springer, 2019.

[12] J. Hedengren and T. Edgar, "Moving horizon estimation-the explicit solution," in *Proceedings of chemical process control (CPC) VII conference. Lake Louise, Alberta, Canada*, 2006.

[13] X. Man and C. Swan, "A mathematical modeling framework for analysis of functional clothing," *Journal of Engineered Fibers and Fabrics*, vol. 2, 11 2007.

[14] F. Coltraro, "Experimental validation of an inextensible cloth model." Technical Report, 2020.

[15] F. Coltraro, J. Amorós, M. Alberich-Carramiñana, and C. Torras, "An Inextensible Model for Robotic Simulations of Textiles." https://arxiv.org/abs/2103.09586, 2021.

[16] D. Parent, A. Colomé, C. Ocampo-Martinez, and C. Torras, "Robotic Cloth Manipulation: Using Model Predictive Control for Cloth State Tracking." Technical Report, 2021.

[17] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, p. 43–54, Association for Computing Machinery, 1998.

[18] Y. Bai, W. Yu, and C. K. Liu, "Dexterous manipulation of cloth," *Computer Graphics Forum*, vol. 35, no. 2, pp. 523–532, 2016.

[19] Z. Erickson, H. M. Clever, G. Turk, C. K. Liu, and C. C. Kemp, "Deep haptic model predictive control for robot-assisted dressing," in *2018 IEEE international conference on robotics and automation (ICRA)*, pp. 4437–4444, IEEE, 2018.

[20] The MathWorks Inc., *MATLAB Version 9.7.0 (R2019b)*. 2019.

[21] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, In Press, 2018.

[22] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, no. 1, pp. 25–57, 2006.

[23] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[24] S. Sarabandi and F. Thomas, "Accurate computation of quaternions from rotation matrices," in *International Symposium on Advances in Robot Kinematics*, pp. 39–46, Springer, 2018.

[25] J. M. Maciejowski, *Predictive control: with constraints*. Pearson education, 2002.

[26] L. Magni, D. Pala, and R. Scattolini, "Stochastic model predictive control of constrained linear systems with additive uncertainty," in *2009 European Control Conference (ECC)*, pp. 2235–2240, IEEE, 2009.

[27] D. Mayne, "Robust and stochastic model predictive control: Are we going in the right direction?," *Annual Reviews in Control*, vol. 41, pp. 184–192, 2016.

[28] P. Velarde, L. Valverde, J. M. Maestre, C. Ocampo-Martínez, and C. Bordons, "On the comparison of stochastic model predictive control strategies applied to a hydrogen-based microgrid," *Journal of Power Sources*, vol. 343, pp. 161–173, 2017.

[29] J. M. Grosso, J. M. Maestre, C. Ocampo-Martinez, and V. Puig, "On the assessment of tree-based and chance-constrained predictive control approaches applied to drinking water networks," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 6240–6245, 2014.

[30] D. C. Montgomery and G. C. Runger, *Applied statistics and probability for engineers*. 2010.

[31] P. I. Corke, *Robotics, Vision & Control: Fundamental Algorithms in MATLAB*. Springer, second ed., 2017. ISBN 978-3-319-54413-7.

[32] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[33] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, learning, and optimization*, vol. 12, no. 3, 2012.

[34] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[35] S. Schaal *et al.*, "Learning from demonstration," *Advances in neural information processing systems*, pp. 1040–1046, 1997.

[36] J. Kober and J. Peters, "Policy search for motor primitives in robotics," *Machine learning*, vol. 84, no. 1-2, pp. 171–203, 2011.

[37] M. P. Deisenroth, G. Neumann, J. Peters, *et al.*, "A survey on policy search for robotics," *Foundations and trends in Robotics*, vol. 2, no. 1-2, pp. 388–403, 2013.

[38] J. Peters, K. Mulling, and Y. Altun, "Relative entropy policy search," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[39] A. Colomé, G. Neumann, J. Peters, and C. Torras, "Dimensionality reduction for probabilistic movement primitives," in *IEEE-RAS International Conference on Humanoid Robots*, 2014.

[40] A. Colomé and C. Torras, "Dual reps: A generalization of relative entropy policy search exploiting bad experiences," *IEEE Transactions on Robotics*, vol. 33, no. 4, pp. 978–985, 2017.

[41] D. Görges, "Relations between model predictive control and reinforcement learning," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 4920–4928, 2017.

[42] M. A. Bermeo-Ayerbe, C. Ocampo-Martínez, and J. Diaz-Rozo, "Adaptive predictive control for peripheral equipment management to enhance energy efficiency in smart manufacturing systems," *Journal of Cleaner Production*, vol. 291, p. 125556, 2021.

[43] J. M. Grosso, C. Ocampo-Martínez, and V. Puig, "Learning-based tuning of supervisory model predictive control for drinking water networks," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 7, pp. 1741–1750, 2013.

[44] N. Karnchanachari, M. I. Valls, D. Hoeller, and M. Hutter, "Practical reinforcement learning for mpc: Learning from sparse objectives in under an hour on a real robot," in *Learning for Dynamics and Control*, pp. 211–224, PMLR, 2020.

[45] Stanford Artificial Intelligence Laboratory, "Robot Operating System - Kinetic Kame." https://wiki.ros.org/kinetic. [Online; accessed July 2021].

[46] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http://eigen.tuxfamily.org, 2010. [Online; accessed July 2021].

[47] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi Documentation." https://web.casadi.org/docs/. [Online; accessed July 2021].

[48] J. Denavit and R. S. Hartenberg, "A kinematic notation for lower-pair mechanisms based on matrices," 1955.

[49] Barrett Technology, Inc., "Barrett WAM 7-DOF Joints & Frames." https://web.barrett.com/files/B2576_RevAC-00.pdf, 2005. [Online; accessed July 2021].

[50] S. G. Khan, G. Herrmann, M. Al Grafi, T. Pipe, and C. Melhuish, "Compliance control and human–robot interaction: Part 1—survey," *International journal of humanoid robotics*, vol. 11, no. 03, p. 1430001, 2014.

[51] S. Harding and J. F. Miller, "Evolution of robot controller using cartesian genetic programming," in *European Conference on Genetic Programming*, pp. 62–73, Springer, 2005.

[52] D. Parent Alonso, "Control d'impedància variable amb primitives de moviment en espais latents amb comportament dòcil," B.S. thesis, Universitat Politècnica de Catalunya, 2019.

[53] D. Parent, A. Colomé, and C. Torras, "Variable impedance control in cartesian latent space while avoiding obstacles in null space," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9888–9894, IEEE, 2020.

[54] M. Arduengo, C. X. Zheng, A. Colomé, and C. Torras, "Cloth Point Cloud Segmentation." https://github.com/MiguelARD/cloth_point_cloud_segmentation, 2021.

[55] S. W. Smith, "Chapter 15 - moving average filters," in *Digital Signal Processing* (S. W. Smith, ed.), pp. 277–284, Boston: Newnes, 2003.

[56] U. P. de Catalunya, "El model docent: Què és un crèdit ECTS?." https://www.upc.edu/ca/graus/faqs/model-docent, 2021. [Online; accessed September 2021].

[57] European Parliament, "DIRECTIVA 2002/95/CE del Parlamento Europeo y del Consejo de 27 de enero de 2003 sobre restricciones a la utilización de determinadas sustancias peligrosas en aparatos

eléctricos y electrónicos.” https://www.boe.es/doue/2003/037/L00019-00023.pdf, 2003. [Online; accessed September 2021].

[58] Hewlett-Packard, Inc., “Workstation ZBook Power G7 Datasheet.” https://www8.hp.com/h20195/V2/GetPDF.aspx/c06908501, 2021. [Online; accessed July 2021].

[59] Barrett Technology, Inc., “WAM Arm Datasheet.” https://web.barrett.com/files/WAMDataSheet_02.2011.pdf, 2011. [Online; accessed September 2021].

[60] Gobierno de España, “Factores de emisión de CO2 y coeficientes de paso a energía primaria de diferentes fuentes de enería final consumidas en el sector de edificios en España.” https://energia.gob.es/desarrollo/EficienciaEnergetica/RITE/Reconocidos/Reconocidos/Otros%20documentos/Factores_emision_CO2.pdf, 2016. [Online; accessed September 2021].