POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Investigation of a Model-Based Approach for Dynamical Manipulation of Deformable Objects

Supervisors

Prof. Massimo CANALE

Prof. Ville KYRKI

Postdoc. Gökhan ALCAN

Candidate

Alessia DE MARCO

October 2023

A nonno Gaetano,
nonna Antonietta,
nonno Mario
e nonna Michielina

Alla mia famiglia,
per avermi lasciata andare
e avermi supportata in ogni mia scelta
anche a 3000 chilometri di distanza.

Acknowledgements

First and foremost, I extend my sincere gratitude to the Intelligent Robotics research group at Aalto University in Finland, with whom this work has been conducted. Their expertise and collaboration have greatly enriched the outcomes of this research.

I am deeply indebted to Prof. Massimo Canale and Prof. Ville Kyrki for their trust in me and for their guidance throughout the completion of this work. Their mentorship and insightful feedback have been instrumental in shaping the final outcome.

I would like to extend a special mention to Gökhan Alcan for his unwavering support and guidance throughout the entire process. His constant advice and assistance have been invaluable, and I am truly grateful for his trust in me.

This thesis represents the culmination of my academic journey over the past two years, which has taken me from Italy, at Politecnico di Torino, to Finland, at Aalto University. Throughout this period, I have experienced tremendous growth both academically and personally, shaping the person I am today.

I extend my heartfelt gratitude to all those who supported me on a daily basis, particularly during my final year. To my family and friends, who stood by me with unwavering commitment despite the distance, and to all those who shared this journey, I am profoundly grateful for your constant support and encouragement. Your unwavering belief in my abilities served as a driving force, motivating me to overcome challenges and pursue excellence.

Summary

Deformable object manipulation is a rapidly evolving field in robotics with applications in various domains such as manufacturing, healthcare, robotics-assisted surgery and rehabilitation. Traditional motion planning algorithms designed for rigid objects are inadequate for deformable objects, necessitating the development of tailored trajectory optimization techniques. This thesis contributes to the field by investigating a model-based technique for deformable object manipulation. The proposed task involves a dynamical movement to be executed by a robotic arm, fixed at the base. This led to assume that the operational space where the end-effector can move is approximated as a sphere, constraining the manipulated mass within this volume. To tackle this task, the chosen method is the iterative Linear Quadratic Regulator (iLQR) one to solve unconstrained trajectory optimization problems in non-rigid object manipulation. However, since the proposed problem is a constrained one, the Augmented Lagrangian and Method of Multipliers technique is employed to handle them. The Mass-Spring-System (MSS) was chosen for modeling the non-rigid object, specifically a rope. This approach involves representing the rope using five nodes connected by three different types of springs and a damping system, all governed by Hooke's law. The three types of springs employed are elastic, shear and flexion. The elastic and shear springs connect adjacent nodes, while the flexion spring connects two diagonal masses. One notable advantage of the implemented model is its differentiability, which is facilitated by the JAX environment. Extensive tuning of the rope model is performed to approximate real-world behavior, and the entire AL-iLQR algorithm is tested and fine-tuned for optimal performance. Parameter variations, including the cost function, horizon length and frequency rate, are analyzed to understand their impact on the controller's behavior. The importance of selecting an appropriate cost function is emphasized through an extensive analysis on the parameters and its shape. The limitations of the Augmented Lagrangian and Method of Multipliers (ALMM) algorithm, particularly the influence of the penalty parameter μ, are deeply discussed. Furthermore, an in-depth analysis is conducted on the horizon length, highlighting its task-dependence and significance in achieving desirable performance. In conclusion, the research highlights the importance of accurate

models, suitable cost functions, and the trade-offs involved in balancing simulation and control aspects. It provides valuable insights and establishes foundations for further advancements in constrained trajectory optimization for non-rigid object manipulation.

Table of Contents

List of Tables

List of Figures

Acronyms

AL-iLQR
Augmented Lagrangian iterative Linear Quadratic Regulator

ALMM
Augmented Lagrangian and Multipliers Method

CDDP
Constrained Differential Dynamic Programming

CGD
Competitive Gradient Descent

DDP
Differential Dynamic Programming

DLO
Deformable Linear Object

FEM
Finite Element Method

HJB
Hamilton-Jacobi-Bellman

iLQR
iterative Linear Quadratic Regulator

KKT
Karush-Kuhn-Tucker

LQR

Linear Quadratic Regulator

MSS

Mass-Spring-System

ODE

Ordinary Differential Equation

PBD

Position Based Dynamics

RK4

Runge-Kutta 4th order method

Chapter 1

Introduction

Deformable object manipulation has become an important area of research in robotics, offering vast potential across various fields. Fabrics, cables, and soft biological tissues are examples of deformable objects with complex dynamics that necessitate specialized techniques for precise and controlled manipulation.
The applications of deformable object manipulation span diverse fields, including manufacturing, healthcare, robotics-assisted surgery, interactive virtual environments, and rehabilitation. In manufacturing, the ability to manipulate deformable materials facilitates automated production processes involving tasks like folding, cutting, and stitching. In healthcare and surgery, precise control over the manipulation of soft tissues is critical for robotic surgical procedures to ensure patient safety and minimize tissue damage. Furthermore, deformable object manipulation has implications in haptic interfaces, virtual reality, and augmented reality, enhancing user interactions and providing realistic simulations.
This research focuses on optimizing and controlling trajectories for dynamically manipulating deformable objects.
Optimizing the trajectory of a robotic manipulator interacting with deformable objects is a crucial task. Traditional motion planning algorithms designed for rigid objects often fail to consider the properties of deformable objects, such as elasticity, compliance, and sensitivity to external forces. Therefore, developing trajectory optimization techniques tailored to deformable objects is essential for achieving accurate and efficient manipulation.
Additionally, controlling the trajectory is equally critical to ensure successful manipulation outcomes. Control algorithms must account for the constantly changing dynamics of the deformable object during interaction, adapting to variations in its shape, material properties, and external forces.
In recent years, research in deformable object manipulation has advanced rapidly, drawing upon multidisciplinary approaches that incorporate robotics, computer vision, machine learning, and materials science. Techniques such as physics-based

simulation, data-driven models, and hybrid control strategies have been explored to address the challenges in this field. Advancements in sensing technologies, including tactile sensing and vision-based sensing, have further enriched the understanding of deformable object behavior and provided valuable feedback for closed-loop control. By addressing the challenges and opportunities in trajectory optimization and control for deformable object manipulation, this research thesis aims to contribute to the growing body of knowledge in the field.

1.1 Research questions and objectives

This thesis aims to evaluate the benefits of using a model-based approach to tackle the task of manipulating deformable objects. Despite the growing interest in model-free approaches, it is important to acknowledge the value of model-based techniques in handling the complexities of deformable objects. These techniques utilize well-defined mathematical models that accurately represent the physical properties and behavior of deformable objects, establishing a strong basis for trajectory planning and control. By emphasizing the advantages of model-based techniques, this research aims to highlight their significance and contributions.
The primary objective of this thesis is to address the aforementioned challenge by employing a model-based approach, specifically a variant of the Differential Dynamical Programming (DDP) algorithm known as iterative Linear Quadratic Regulator (iLQR), to solve a constrained trajectory optimization problem.
Additionally, the thesis aims to explore two related objectives: how to handle constraints and which model to employ for describing deformable objects. To address the first objective, a relaxation technique based on the Augmented Lagrangian method will be applied. In terms of modeling the non-rigid object, a physics-based method known as Mass-Spring-System will be implemented due to its efficiency, simplicity and intuitiveness. An important feature of the model is its differentiability, which is highly advantageous.
By pursuing these research objectives, the thesis aims to contribute to the field of deformable object manipulation and shed light on the advantages of model-based approaches in terms of accuracy, precision, and computational efficiency compared to model-free alternatives.

1.1.1 Description of the task

The specific task of this thesis project is to investigate the properties and the behavior of the chosen model-based approach for a dynamical manipulation task. The manipulated object is a piece of rope, held by one of its ends by the end-effector of a robotic arm. The task involves a dynamical movement of the end-effector to

bring the non-rigid object from the starting position shown in Figure 1.1 to the goal position shown in Figure 1.2.

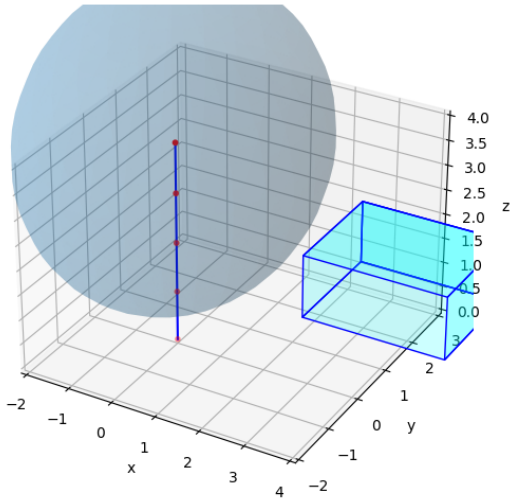

Figure 1.1: Starting configuration

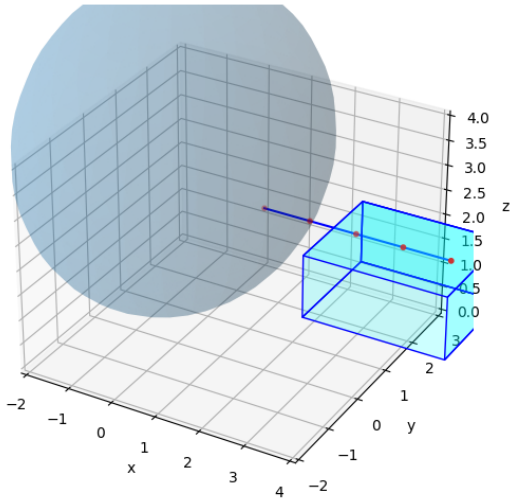

Figure 1.2: Goal configuration

During the execution of the movement, the manipulated mass, the one in the center of the sphere, must remain inside the sphere. The box, where the rope is to be laid as the target position, is designed in such a way that the rope cannot pass through it.
The objective is to generate a trajectory that achieves the needed dynamical impulse to fulfill the goal.

1.2 Structure of the thesis

The structure of this thesis is organized as follows. Chapter 2 provides an exploration of the relevant literature pertaining to the main topic. It also introduces the necessary mathematical tools and theory of optimization and optimal control. These foundations will serve as a basis for understanding the employed approach and methodology to address the proposed problem, which will be detailed in Chapter 3.
Chapter 3 focuses on presenting the utilized approach and the applied methodology. It will delve into the specific algorithm employed to tackle the problem at hand.
Moving on to Chapter 4, the performance of the algorithm will be thoroughly analyzed and discussed. This chapter aims to provide a comprehensive evaluation of the algorithm's effectiveness.

Lastly, Chapter 5 summarizes the relevant findings and contributions of this work. It will provide a concise overview of the results and conclude the thesis by highlighting its significance in the context of the research field.

Chapter 2

Literature Review and Background

In this chapter, the literature about optimal control and trajectory optimization for manipulation of deformable objects will be explored, along with an overview of the needed theoretical tools to understand the approach to the solution for the problem presented in the Introduction. In particular, in the first part, the literature about modeling and control technique for manipulation of non-rigid objects will be reviewed, while, in the second part, an overview on **optimization** and **control theory** needed for model-based approaches will be given.

2.1 Literature review

In order to achieve the dexterous properties of the human body, robots need to fulfill the task of handling non-rigid objects. In the last years, this challenge has been explored, even though the same dexterity as with the manipulation of rigid objects has not been achieved.
There are various surveys that collect the work done up to now in this field, like the one presented by Jiménez [1], which includes an overview of modelling and control techniques, or the one of Nadon et al. [2] which focuses more on 3D non-rigid objects manipulation and control. All these approaches share a common objective of describing non-rigid objects. Typically, they involve modeling the object's shape over time by combining a representation for the object's surface with the associated deformations. However, there are also solutions that do not rely on a predefined model. Instead, they acquire the necessary information about the deformation and shape of the non-rigid object through the use of sensors.
The first scenario is known as *model-based* approach, whilst the latter as *model-free* approach. The choice of the best option has to be made considering the task itself,

the computational resources available and the accuracy that has to be achieved. In the paragraphs below, the main differences between these two approaches will be highlighted along with the different strategies that can be used in both situations in order to evaluate the best approach to use.

2.1.1 Model-based approaches

A *model-based* approach for manipulation and control of objects, as the name says, makes use of a model to describe the dynamics of the object involved in the task. Once the model is established, this approach can be used to plan and optimize actions for manipulation tasks; in this sense, it refers to trajectory optimization and control tools in order to solve variations of the following problem in discrete-time where the model of the non-rigid object is needed for the equation of the system's dynamics: [3]

$$\begin{aligned} \min_{x_{0:N}, u_{0:N-1}, \Delta t} \quad & \ell_N(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k, \Delta t) \\ \text{subject to:} \quad & x_{k+1} = f(x_k, u_k, \Delta t), \\ & g_k(x_k, u_k) \leq 0, \\ & h_k(x_k, u_k) = 0 \end{aligned} \tag{2.1}$$

Here, k is the time-step index, ℓ_N and ℓ_k are the terminal and stage costs, x_k and u_k are the states and control inputs, Δt is the duration of a time-step, $f(x_k, u_k, \Delta t)$ is the discrete dynamics and $g_k(x_k, u_k)$ and $h_k(x_k, u_k)$ are inequality and equality constraints.

To tackle such problems, two types of solvers exist: *direct* and *indirect* methods. Direct methods approach the problem by considering both states and controls as decision variables. They make use of widely-used general-purpose nonlinear programming solvers (NLP) such as SNOPT and IPOPT. These solvers are well-known for their numerical robustness and versatility, allowing them to effectively handle various types of optimization problems, including those with diverse constraints. However, the optimization process is slowed down because feasibility of dynamics needs to be imposed.

In contrast, indirect methods leverage the Markov structure of the problems and only treat the control inputs as decision variables. The dynamics constraints are implicitly enforced by simulating the system's dynamics. This approach leads to faster computation.

Differential Dynamic Programming (DDP), initially introduced by Jacobson and Mayne [4], is recognized as one of the most successful trajectory optimization algorithms. Unlike classical approaches like Direct Collocation or Direct Multiple

Shooting, which optimize open-loop trajectories, DDP and its variants take a different approach. They simultaneously design both a stabilizing feedback controller and a feedforward controller.
These iterative algorithms are specifically designed to solve optimal control problems that involve nonlinear cost functions and nonlinear system dynamics. By iteratively updating the controllers and refining the trajectory, DDP provides an efficient solution to these challenging problems.
The main reason of its popularity is that often the computational complexity is linear in the number of time steps, $O(N)$ [Giftthaler et al. 5], and, moreover, under some assumptions it can shown *quadratic convergence* in unconstrained problems [Liao and Shoemaker 6]. Murray and Yakowitz [7] conducted a comparison between Newton's method and DDP for solving *unconstrained problems*. Their findings demonstrated that although DDP and Newton's method are not identical, they exhibit similar convergence rates, specifically quadratic convergence.
However, in the current work, the optimization problem at hand involves *constraints*. Therefore, it is important to consider this aspect in the analysis and solution.
A specific version of DDP tailored for constrained optimal control, applied to a multireservoir control problem [8], relies on the "stagewise" Kuhn-Tucker condition [9]. The results of this study suggest that DDP is the most effective technique among the available methods for a certain class of large-scale control problems. This research highlighted the remarkable power, robustness, and reliability of DDP when second derivatives can be conveniently calculated. Consequently, for the majority of optimal control problems, DDP is considered the method of choice.
Box inequality constraints on control inputs can be included in DDP without significantly sacrificing convergence quality or computational effort, using the method of Tassa et al. [10]. However, the method does not apply to state constraints.
Augmented Lagrangian methods offer an alternative approach for handling constraints, and they can be seamlessly integrated into DDP. Aoyama et al. [11] demonstrated how a specific set of penalty-Lagrangian functions, which maintain second-order differentiability, can be effectively incorporated within DDP.
Lin and Arora [12] proposed two computational procedures based on DDP for solving linear or nonlinear constrained optimal control problems.
The first procedure involves solving a *quadratic programming problem* at each time step, allowing the construction of a differential control law. This approach enables handling point-wise constraints effectively.
The second procedure, known as the *multiplier method*, aggregates all the pointwise constraints and employs the unconstrained DDP approach to construct the differential control law. The advantage of this method over the first one is that it imposes no limitations on the constraints, making it more general and flexible.
Xie et al. [13] introduced a Constrained version of DDP that addressed the incorporation of nonlinear constraints. In their work, they focused on deriving a recursive

quadratic approximation formula for the optimization problem while considering the presence of nonlinear constraints. This was accomplished by identifying a set of active constraints at each point in time.
Their CDDP algorithm is capable of iteratively determining the active set, guaranteeing convergence to a local minimum. It has been successfully applied to underactuated optimal control problems, including those with up to 12 dimensions, involving obstacle avoidance and control constraints. CDDP has demonstrated superior performance compared to other methods in handling constraints.
During the comparative analysis, CDDP was evaluated against two alternative methods. The first method replaces hard constraints with a log-barrier penalty term incorporated into the objective function, while the second method utilizes sequential quadratic programming implemented through SNOPT software.
CDDP demonstrates superior performance in longer horizons compared to SNOPT. Furthermore, when compared to the log-barrier constraint penalty approach, CDDP converges in fewer iterations and exhibits better capabilities in handling more complex dynamic problems.
However, it is important to note that the implementation of CDDP is relatively slower than the standard DDP method. This is due to the need to solve quadratic programs instead of employing matrix inversion in the inner steps of the algorithm. Additionally, like any nonconvex optimization problem, a well-defined initial trajectory is crucial to prevent the algorithm from getting trapped in a bad local minima. Finally, while the original DDP relies on second-order derivatives, one of its variations, *iterative-Linear-Quadratic-Regulator* (iLQR), uses only Gauss-Newton approximations of the cost Hessians as well as first-order expansions of the dynamics: Li and Todorov [14] proved that this simplified version can be faster in time and iterations with respect to DDP and other methods like ODE, which solves the system of state-costate ordinary differential equations resulting from Maximum Principle, and CGD, that is a gradient descent method based on the Maximum Principle to compute the gradient of the total cost with respect to the nominal control sequence, and then calls an optimized conjugate gradient descent routine.
To summarise the literature on DDP, its efficiency in tackling multiple optimisation problems has been amply demonstrated. Above all, its flexibility in adapting the method to constrained problems by combining it with others that can handle the constraints without losing convergence quality has been highlighted. Lastly, the simplified version, iLQR, which allows for fewer iterations and savings in computational resources, should be emphasised.

Modeling the deformable object

A model-based approach requires a simulation model that captures the dynamics of the system.

Manipulation planning focuses on displacing objects to be manipulated, rather than solely planning the robot's motion as in standard motion planning. Manipulating a rigid object involves changing its pose, which refers to altering its position and orientation while avoiding collisions. This type of manipulation is typically used for tasks like pick-and-place or assembly.
When dealing with deformable objects, manipulation also affects their shape, resulting in geometric or topological changes. These shape changes must be considered throughout the planning process.
Planning involves selecting or generating a series of actions from a set of options. The decision-making process considers not only the desired outcomes, such as the arrangement of manipulated objects, but also the constraints imposed by each step in the sequence. These constraints can include both internal and external limitations, as well as feasibility and optimization considerations, which can be classified as hard and soft constraints, respectively. By implementing constraints, the range of potential actions is narrowed, and criteria for evaluating and comparing different choices are established.
The deformations of an object are influenced by various factors, including the material properties of the objects, their initial shape and dimensions, and the specific application of forces for manipulation, such as localization, direction, intensity, duration, and frequency. Two key concepts in object deformations and manipulation are the reversibility of deformation and the direction and extent of shape change. In the case of reversibility, the object fully recovers its original shape, known as the rest shape, when external forces are ceased, and this is referred to as *elastic* deformation. On the other hand, permanent and stable changes result in *plastic* deformation.
A third category, called *flexible* deformations, applies to objects that cannot be strictly classified as either elastic or plastic due to the effects of friction and gravity, preventing them from returning to a true "rest shape" after manipulation. However, these objects can still be altered with minimal effort, indicating that they do not exhibit strictly plastic properties. Examples include materials like rope or cloth.
Furthermore, deformations are also characterized by the direction in which they occur, determined by the applied forces and moments. Following this, the aim of models is to replicate the behavior of manipulated objects, and the accuracy of these models is dependent on the purpose. Furthermore, the trade-off between computational efficiency and precision is a crucial factor to consider.
Models aim to replicate the behavior of manipulated objects, and the accuracy of these models depends on their intended purpose. There is a trade-off between computational efficiency and precision that needs to be considered.
In contrast to rigid objects, collisions are often less problematic during the manipulation of deformable objects because their shape can adapt to obstacles. Therefore,

assumptions about the shape of deformable objects can be simplified. Since deformable objects change their shape in response to internal and external forces, this can be either a desired goal of the planning process, such as when they help avoid obstacles along a path, or a side effect of a planned action.
In any case, the changes in shape are governed by the underlying physical behavior of the object in three-dimensional space. This behavior can be implicitly considered using pure geometric models or explicitly incorporated into the model.
As mentioned earlier, the DDP algorithm and its iLQR variant rely on derivatives of the dynamics equation. Having a **differentiable** model for the object can be beneficial as it allows the use of built-in functions for gradient retrieval. However, it is not always possible to obtain a closed-form solution for the derivatives. Fortunately, this issue can be addressed by employing an implicit integration scheme, which enables the computation of gradients and Hessians.
Indeed, the challenges inherent to dynamic manipulation of non-rigid items, according to Zimmermann et al. [15], arise due to the need to employ advanced constitutive material models, such as FEM, that relate deformations to internal restorative forces. Furthermore, these suffer from numerical stiffness issues, which can only be stably handled by applying *implicit* integration schemes. Conversely, explicit schemes tend to be unstable and blow-up quickly, making them practically unusable.
Their research focused on adapting Newton's method and DDP to physical systems that are simulated forward in time using implicit integration schemes. Both methods are classified as "single shooting" methods, but they differ in their approach. The first method operates on the entire control sequence simultaneously, while the latter is a stage-wise method that processes the control sequence recursively. Implicit integration methods, unlike explicit alternatives, do not have closed-form solutions available at each time step. To overcome this challenge, the researchers demonstrated how sensitivity analysis can be utilized to analytically compute the derivatives required by both methods.
The effectiveness of their solutions was evaluated through experiments involving dynamic motions of an elastic object. To assess the trajectory optimization formulations presented, they utilized a dual-armed YuMi IRB 14000 robot and various types of foam-based elastic objects. The model of the elastic objects employed the FEM, and the numerical integration scheme used was the second-order accurate Backward Differentiation Formula (BDF2).
By incorporating suitable regularizers and employing line search mechanisms, both DDP and Newton's method demonstrated reliable convergence. DDP exhibited better scalability than Newton's method in terms of the planning horizon length. The overall computational cost heavily depends on the dimensionality of the object's state space and its sparsity structure. The relative performance of the methods appeared to be highly dependent on the specific nature of the problem.

Additionally, the performances of the proposed methods were compared to an alternative trajectory optimization approach known as the Direct Collocation method, which solves the problem through constrained optimization. Direct Collocation, using techniques like the interior point method, allows constraints to be violated during optimization, potentially speeding up the process. However, it typically requires more iterations compared to DDP and Newton's method.
In general, the use of explicit or implicit integration schemes depends on the type of model for the deformable objects.
The deformable modelling strategies can be categorized following a previous model categorization presented by Montagnat et al. [16] and Salzmann and Fua [17] and they are: physical, geometric, learned and hybrid models.

Physical models

Physical models aim to replicate the behavior of objects based on the principles of physical laws. Two prominent examples in this category are Mass-Spring System (MSS) simulations and Finite Element Method (FEM). The fundamental concept behind these models is to account for the fact that deformable objects possess infinite degrees of freedom by discretizing the structure, thereby reducing the degrees of freedom to a finite number.
Both of them are based on Newton's law of motion, under which particle or vertex motion can be described by time derivative of momentum and exerted forces f as

$$M\ddot{p} = f \tag{2.2}$$

where M and p denote the system mass and position, respectively.
The motion of the entire system can be computed by integrating from an initial state x_0. For each particle, the evolution is defined as:

$$\begin{aligned} \dot{p_i} &= v_i \\ \dot{v_i} &= \frac{1}{m_i}(f_i^{int} + f_i^{ext}) \end{aligned} \tag{2.3}$$

with $\dot{p_i} = v_i$, that represents the velocity of the i^{th} particle.
The exerted forces explicitly divided into two parts: the component f_i^{ext} sums up *external contributions*, such as gravity and input forces, which are known for the given time step; whereas the term f_i^{int} has the important role of resuming the internal effects that characterize the *deformations* and the non-rigid object itself.
After resolving the force terms, simulating the system using explicit Euler integration or other methods such as Runge-Kutta4 (RK4) is a straightforward process. The integration scheme chosen depends not only on its computational efficiency but also on the stability of the dynamics being simulated.

Generally, FEM-based models are widely recognized as the most popular category of physical models, offering a well-established methodology for formally describing deformations [2].
In literature, there are several works that describe its functioning behaviour, most of them rely on tetrahedral shape and the elasticity properties are describe through *Young's modulus* and *Poisson ratio*. In general, they are *less intuitive* and *more computational expensive* than Mass-Spring Systems, that is a chain of nodes linked by damped springs subjected to Damped-Hooke's law.
However, MSS techniques have widely and effectively been used for modeling non-rigid object because they are faster and easier to compute with a more intuitive physical meaning. Conversely, they have stability issues with high stiffness materials because they require a small time integration step during the simulation process, slowing down the entire process [18].
Additionally, physical models rely on material parameters, which are often unknown or challenging to describe accurately in a broader context, particularly for objects with complex material structures (Yin et al. [19]).

Geometric Models

Geometric models are approaches that focus on representing the surface and its evolution over time purely as a geometric model, without considering physical information. One example of such a model is the *Position-Based Dynamics* (PBD) model. PBD is a mesh-free method that represents materials as a discrete system composed of particles. The simulation in PBD follows an implicit integration scheme, where internal forces are derived from holonomic constraints, including temporary and unilateral constraints.
Geometric models offer the advantage of accurately depicting the complete deformation of an object while effectively preserving its shape. Additionally, PBD provides a fast and well-controlled simulation with improved stability. This method allows for the easy incorporation of various constraints and provides the ability to guide the system by setting specific boundary conditions.
However, one limitation of geometric models like PBD is that it is not always straightforward to assign a meaningful physical interpretation to some of the parameters used. Tuning efforts may be required to achieve desired effects compared to physical models. (Yin et al.[19]).
Other drawbacks are its limited accuracy in simulating force effects, and that its behavior is dependent on the time step and iteration count of the simulation (Bender et al. [20]), to solve this Macklin et al.[21] introduced their extended-Position-Based Dynamics model (XPBD) that has a new constraint formulation which corresponds to a well-defined concept of elastic potential energy.
Liu et al. [22], instead, introduced a compliant position-based dynamics to model

rope-like objects, through the use of geometric constraints which can reproduce shear/stretch and bend/twist effects. One important feature of their work was the *differentiability* of the model which improves the functioning behaviour of the model and simulation environment.
In the following Table, the considerations are summarized with pros and cons and respective application fields.

	Advantages	**Limitations**	**Modeling applications**
MSSs	Fast	Inaccurate for large deformation	Ropes
			Fabrics
	Simple to implement	Lacking physical interpretability	Sponges
			Rubber spheres
PBD	Fast and stable	Visual fidelity only	Paper
			Fabrics
	Supports modeling of various objects	Lacking physical interpretability	Cushion
			Fabrics
FEMs	Can be fast (linear)	Complex and expensive to compute (nonlinear)	Rods and cables
			Fabrics
	Physical fidelity and interpretability	Not well integrated to robotics simulators	Food
			Tissues

Table 2.1: Overview of deformable objects modeling approaches (Yin et al. [19])

Learned models and hybrid models

The last two categorizations are *learned* and *hybrid* models.
Learned models use the available information as training data to infer the shape of the object; in this way it is addressed the challenge of representing non-linear objects and estimating unknown parameters.
A wide used tool according to the literature is Neural Network (NN); they are trained based on data collected from available sensors, such as images, and the outputs represents the interested object with its deformations [23].
Hybrid models, instead, combine two or more of the previous modelling methods to improve performance and accuracy [24].

2.1.2 Model-free approaches

The *model-free* approach to manipulating non-rigid objects involves utilizing machine learning techniques to learn control policies or behaviors directly, without

explicitly constructing a model of the object's dynamics.
In this approach, data is collected through trial and error or demonstrations, and a policy is trained to replicate the observed behavior. Visual tools are used to extract various parameters of the object, including shape, dimensions, position, and orientation. These parameters are then used to search for the best fit in a database. This approach proves particularly valuable when accurately modeling the complex physics of non-rigid objects is challenging. In fact, it employs methodologies that enable online estimation of the deformations.
Model-free methods, such as reinforcement learning, empower a robot to adapt and learn from its interactions with the object. This adaptability allows for robust manipulation, even in the presence of uncertainties or variations in object properties. For example, Chi et al.[25] proposed the *Iterative Residual Policy* (IRP), that is a *learning framework* applicable to repeatable tasks with complex dynamics and high-speed actions with a strict requirements on the task to be achieved, such as, in that case, hitting a target positioned somewhere in the operational space with a rope. It learns an implicit policy via delta dynamics instead of modeling the whole dynamical system and inferring actions from that model; moreover, it predicts the effects of delta action on the *previously-observed trajectory*. Hence, it uses a *visual feedback* to improve the action and the trajectory. Machine learning has proven to be highly effective in dealing with complex interactions when controlling non-rigid objects. This is especially beneficial in scenarios where it is challenging to develop precise control algorithms due to the complexity of task descriptions or capturing all relevant parameters.
Among the different learning approaches for robotic manipulation of non-rigid objects, *learning by demonstration* has emerged as a popular choice. In this paradigm, the robot learns and generalizes tasks by observing a human expert performing them [26], [27] or by imitating human interaction partners [28]. This approach empowers a versatile robot to execute multiple tasks involving non-rigid objects without requiring the development of distinct controllers for each task.
However, while the use of data-driven approaches has gained popularity in the search for controllers that can learn to perform specific tasks, there is still a strong inclination towards developing a model-based method that can address multiple optimization problems.
Model-free approaches indeed may lack interpretability and understanding of the underlying physical principles governing deformable object behavior. This can make it challenging to diagnose and correct errors or unexpected behavior during manipulation, while a model-based approach can address this better because of the knowledge of the physical model, leading to more stability and robustness.
Moreover, due to the fact that using model-free approaches require more computational resources and higher costs because it is required to train a large amount of data to extract the needed features.

It should also be noticed that in order to have accurate predictions of how the object will deform and respond to external forces, leading to more precise manipulation, with a model-based approach, it is required a model that addresses the physical properties and behavior of the deformable objects. The use of accurate models indeed enables optimization and planning techniques to be applied. By formulating the manipulation problem as an optimization or planning task, it becomes possible to find optimal control strategies that minimize certain objectives (e.g., energy consumption, deformation, or contact forces) or satisfy specific constraints. This can lead to more efficient and effective manipulation of deformable objects.
Here, for simplicity, using a 1D deformable objects, like rope, would improve the overall performance, since it has been widely studied and modelled through different techniques.

2.2 Optimization theory

A model-based approach makes use of trajectory optimization and control tools, as already said in Section 2.1.1. Thus here, the main theoretical aspects will be given. *Optimization* allows to find values for variables within a given domain that minimize or maximize the value of a function, i.e. it allows to solve the following general problem:

$$\begin{aligned} &\min f(x) \\ &\text{s.t.: } x \in S \end{aligned} \tag{2.4}$$

This problem can be solved by employing the following strategy:

1. Analyse properties of the function under the specific domains and derive the conditions that must be satisfied such that x is a candidate optimal point;
2. Apply numerical methods that iteratively search for points satisfying these conditions.

In this sense, *variables* represent *decisions*, *domains* are *constraints* that must not be violated and the *function* is an *objective function* which provides a measure of a solution quality.
The general form of a problem is:

$$\begin{aligned} &\min f(x) \\ &\text{s.t.: } g_i(x) \leq 0, i = 1, ..., m \\ &\quad\;\; h_i(x) = 0, i = 1, ..., l \\ &\quad\;\; x \in S \end{aligned} \tag{2.5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the **objective function**, $g : \mathbb{R}^n \to \mathbb{R}^m$ is a set of m **inequality constraints** and $h : \mathbb{R}^n \to \mathbb{R}^l$ is a collection of l **equality constraints**. [29]

Optimality conditions for unconstrained problems

Most optimization methods are based on successively obtaining *directions* d of potential improvement and suitable *step sizes* λ in this direction, until a *convergence or termination criterion* is satisfied, as shown in Algorithm 1. [29]
The problem of the form 2.5 is defined as **unconstrained** if $l = m = 0$, i.e. there aren't equality and inequality constraints, and the set S is the whole $\mathbb{R}^n$.
For unconstrained problem, we can define the **descent direction d** in the following way.
Suppose f is differentiable at $\bar{x}$*. If there is d such that* $\nabla f(\bar{x})^T d < 0$*, there exists* $\delta > 0$ *such that* $f(\bar{x}+\lambda d) < f(\bar{x})$ *for each* $\lambda \in (0, \delta)$*, so that d is a descent direction of f at* $\bar{x}$*.*
This leads to the *first and second order necessary condition.*
The first states that if the function is differentiable and $\bar{x}$ is a local minimum, then $\nabla f(\bar{x}) = 0$.
The second states that if the function is twice differentiable and $\bar{x}$ is a local minimum, then $H(\bar{x})$ is positive semidefinite (PSD).

Algorithm 1 Optimization algorithm

1: ▷ Initialize iteration count $k = 0$, starting point x_0
2: **while** stopping criteria are not met **do**
3: ▷ compute direction d_k
4: ▷ compute step size λ_k
5: $x_{k+1} = x_k + \lambda_k d_k$
6: k=k+1
7: **end while**
8: **return** x_k

Line search and convergence criterion

To improve the efficiency of Algorithm 1, it is needed to choose a *suitable step size* and to *terminate the execution* when the result is accurate enough.
To find a suitable step size is itself an optimization problem referred as **line search**, that consists of a uni-dimensional search as $\lambda_k \in \mathbb{R}$. Hence it is possible to define the function $\theta(\lambda) = f(x+\lambda d)$ and minimizing its value with respect to the variables λ.
There are designed methods to find the optimal value of the step size, called as *exact line search methods*, but often it is worth to sacrifice the optimality of the solution for the efficiency of the whole method in terms of computational time, as the **Armijo rule** that is used in this work.
The Armijo rule is a condition that is tested to decide whether a current step size

$\bar{\lambda}$ is acceptable or not. The step size is considered acceptable if

$$f(x + d\bar{\lambda}) - f(x) \leq \alpha\bar{\lambda}\nabla f(x)^T d \tag{2.6}$$

where α is a predefined constant.
If $\bar{\lambda}$ doesn't satisfy this condition, it is reduced by a term $\beta \in (0,1)$ until the test is satisfied.
This rule is also called *backtracking* due to the successive reduction of the step size by the factor β.
The **convergence** instead is analysed by means of the *rate of convergence* associated with the *error functions* $e : \mathbb{R}^n \to \mathbb{R}$ such that $e(x) \geq 0$. Typical choices are:

- $e(x) = ||x - \bar{x}||$;
- $e(x) = |f(x) - f(\bar{x})|$.

Thus, the algorithm will stop when the chosen error function goes under a defined tolerance or the number of maximum iteration is achieved.[29]

2.2.1 Constrained optimization problems

When dealing with constrained optimization problems of the general form of 2.5, the standard optimization approach of the form shown in the algorithm 1 doesn't work because it doesn't take into account the constraints; hence, it can be modified in order to guarantee that the computed descent direction d leads to a solution that is included in the set S of feasible directions. In this sense, the value $\bar{x}$ is optimal if there exists no feasible direction that can provide improvement in the objective function value.
There are several approaches to handle constraints in optimization problems. One common method is to incorporate constraints directly into the objective function through penalty or barrier functions [11], where violating constraints leads to increased costs. This encourages the optimization algorithm to find solutions that satisfy the constraints.
Another approach is to use inequality constraints, defining feasible regions in the solution space. Optimization algorithms can then search within these regions to find optimal solutions that meet the constraints.
Additionally, constraint handling techniques such as constraint aggregation, constraint relaxation, or constraint transformation can be employed to simplify or reformulate the problem, making it more amenable to optimization algorithms.
Furthermore, advanced optimization techniques like Lagrange multipliers [12], interior-point methods [9], or active-set methods [13] can be utilized to explicitly account for constraints and find solutions that satisfy them. By employing these techniques, constraints can be effectively addressed in optimization, enabling the

discovery of optimal solutions while meeting the desired constraints.
More in details:

- The **interior point method** is an optimization technique used to solve constrained optimization problems. It focuses on finding solutions that satisfy both equality and inequality constraints. This method transforms the original problem into an unconstrained problem by introducing a barrier or penalty function that penalizes violations of the constraints. The method then seeks a solution by iteratively moving towards the interior of the feasible region while maintaining feasibility. This is achieved by updating a set of iterates that satisfy the Karush-Kuhn-Tucker (KKT) optimality conditions. The interior point method is known for its ability to handle large-scale optimization problems efficiently and to find both feasible and near-optimal solutions.

- The **Lagrangian method**, also known as the method of Lagrange multipliers, is a mathematical optimization technique used to solve constrained optimization problems. It involves introducing Lagrange multipliers, which are scalar variables associated with each constraint, to create a new function called the Lagrangian. The Lagrangian combines the objective function with a weighted sum of the constraints, where the weights are the Lagrange multipliers. The Lagrangian method then seeks to find critical points of the Lagrangian by taking partial derivatives with respect to the variables and the Lagrange multipliers. This results in a set of equations called the KKT optimality conditions, which must be satisfied by the optimal solution. Solving these conditions yields the optimal values of the variables and the Lagrange multipliers, providing a solution that satisfies the constraints.

- The **active set method** is an iterative optimization technique used to solve constrained optimization problems with both equality and inequality constraints. It focuses on identifying and updating an active set, which consists of the active constraints that are binding at the current solution. The method starts with an initial guess of the active set and iteratively updates it based on the optimality conditions and feasibility of the solution. In each iteration, the active set is refined by checking for violated constraints and adding or removing them accordingly. The active set method then solves a subproblem restricted to the active set, typically using techniques like quadratic programming or linear programming. This process continues until an optimal solution is found that satisfies the constraints and optimality conditions.

2.3 Control Theory

In order to achieve the proposed task, it is needed to compute the input value, i.e. the external force, that the robotic arm has to apply to the rope to reach the target. We need to define the *controller* that, given the desired state to reach, provides the input values to the system, the end of the rope so that this goal state can be reached as fast as possible with precision and ensured robustness. The block diagram scheme is shown in the Figure 2.1.

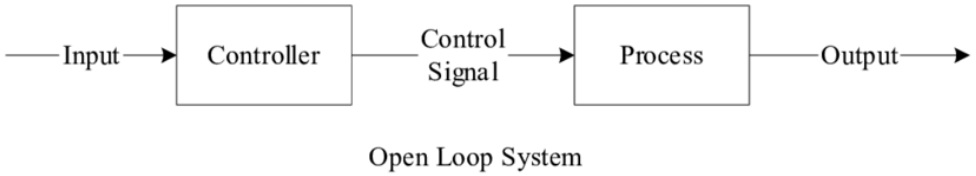

Figure 2.1: Open-loop Control diagram [30]

This is an **open-loop** control scheme because the controller doesn't depend on the new system output, but only on the desired state of the system $x(t)$.
Whereas, if the controller is fed with the difference between the current state and the desired state, i.e. the **error**, it is defined as **closed-loop** and it is more robust against perturbations of the state, as shown in Figure 2.2.

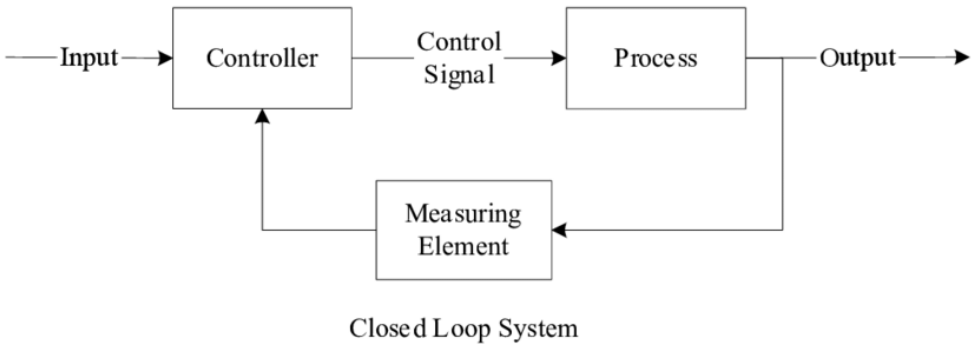

Figure 2.2: Closed-loop Control diagram [30]

2.3.1 Optimal Control

Optimal control is a sub-branch of the control theory, that focuses on minimizing a scalar cost that can be computed at each time-step constrained to the system dynamic of the form $\dot{x} = f(x(t), u(t))$, where $x(t)$ is the state of the system and

$u(t)$ is the input value.
An optimal control problem should then be formulated as:

$$\begin{aligned} &\min_{x,u} \int_0^T \ell(x(t), u(t))dt \\ &s.t. \dot{x(t)} = f(x(t), u(t)) \end{aligned} \tag{2.7}$$

where $\ell()$ is the cost-to-go and T can be either finite or infinite following that in the first case is a "finite-horizon" problem, while in the latter is an "infinite-horizon" problem. Hence, optimal control is a control design process that uses optimization. Due to its generality, it can be applied to a wide range of problems, fully or underactuated, linear or nonlinear, deterministic or stochastic, continuous or discrete systems.
Furthermore, it allows to describe very complex desired behaviour through a scalar objective and a list of constraints and it is amenable to numerical solutions, like **Dynamic Programming** for solving multi-stage decision making problems.

Dynamic Programming

Dynamic Programming (DP) is a method to solve optimal control problems. The key behind this method is to discretize the state space and the control space, to derive the cost-to-go for all the states and then, starting from the final state, to go backwards until the initial state is reached.
The goal is to design some control law u to follow a state trajectory x(t) to minimize a cost function J defined as:

$$J(x(t), u(t), t_0, t_f) = \ell_N(x(t_f), t_f) + \int_0^{t_f} \ell(x(\tau), u(\tau))d\tau \tag{2.8}$$

which is the sum of the running cost and the cost of the final trajectory.
Then, we define the **value function** as:

$$V(x(t_0), t_0, t_f) = \min_{u(t)} J(x(t), u(t), t_0, t_f) \tag{2.9}$$

that means the control law out of all the possible ones that minimizes the cost function.
As said previously, one of the main concepts is to optimize the problem by breaking it into smaller recursive sub-problems. This is due to the Bellman optimality equation:

$$V(x(t_0), t_0, t_f) = V(x(t_0), t_0, t) + V(x(t), t, t_f) \tag{2.10}$$

which expresses the fact that the optimal cost-to-go is the sum of the optimal ones from the initial state to any middle state and from this one to the goal state.

This implies that any point of the trajectory has to be optimal because we are considering it as the initial condition for the remaining path. That's the *Markov decision process*. This concept allows to break the problem into smaller ones and solve them.

We then define the dynamics of the system as:

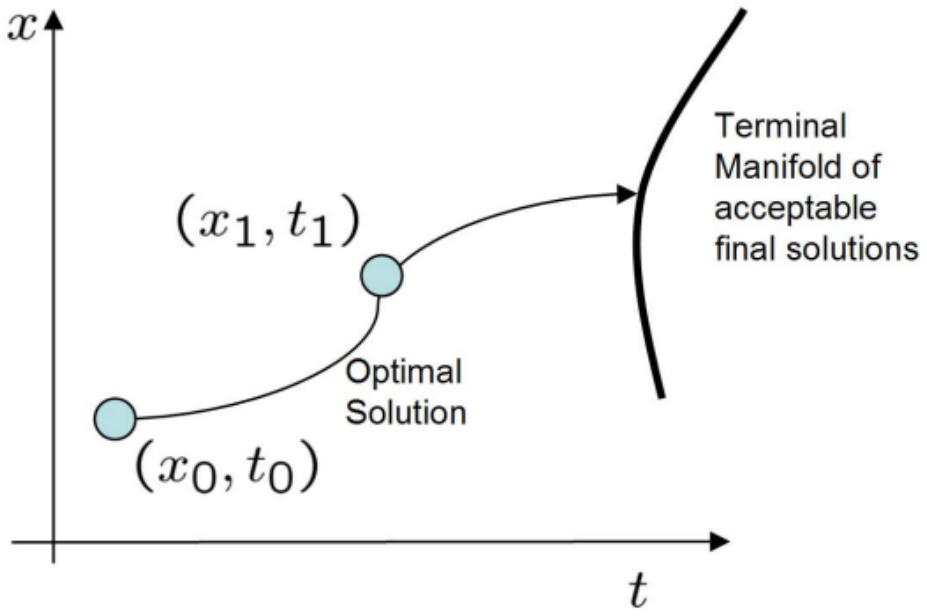

Figure 2.3: Bellman optimality

$$\dot{x} = f(x(t), u(t), t) \tag{2.11}$$

From (2.8) and (2.9), it is possible to derive the sufficient condition for optimality: if it can be found an optimal J^* and V^*, that satisfy those equations, then V^* must be an optimal controller.

For a system $\dot{x} = f(x, u)$ and a horizon additive cost $\int_0^{t_f} \ell(x, u)dt$:

$$0 = \min_u[\ell(x, u) + \frac{\partial J^*}{\partial x} f(x, u)] \tag{2.12}$$

$$\pi^*(x) = arg\min_u[\ell(x, u) + \frac{\partial J^*}{\partial x} f(x, u)] \tag{2.13}$$

Equation 2.13 is the optimal control policy.

Equation 2.12 is known as the **Hamilton-Jacobi-Bellman** equation and can be written also as:

$$-\frac{\partial V}{\partial t} = \min_{u(t)}((\frac{\partial V}{\partial x})^T f(x(t), u(t)) + \ell(x(t), u(t))) \tag{2.14}$$

DP works well when there is a low dimensionality for the control input. It is reliable and it ends up with optimal trajectory 100% of the time. It also allows to include any kind of constraints.
However, evaluating the HJB for the time-to-go reavels the necessity to have a well defined $\frac{\partial J}{\partial x}$ everywhere. [31]

Linear Quadratic Regulators

In general, solving the dynamic programming problem for continuous systems is very hard, but there are few special cases where the solutions are accessible, like variants that involves linear dynamics and quadratic cost. The simplest is the linear quadratic regulator (LQR), formulated as stabilizing a time-invariant linear system to the origin. [31]

Fundamentals

Consider a linear time-invariant (LTI) system in state-space form,

$$\dot{x} = Ax + Bu, \tag{2.15}$$

with the infinite-horizon cost function given by

$$J = \int_0^\infty [x^T Q x + u^T R u] dt, \quad Q = Q^T \succeq 0,\ R = R^T \succ 0 \tag{2.16}$$

The goal is to find the optimal cost-to-go function $J^*(x)$ that satisfies the HJB:

$$\forall x, \quad 0 = \min_u [x^T Q x + u^T R u + \frac{\partial J^*}{\partial x}(Ax + Bu)] \tag{2.17}$$

To verify that the optimal cost-to-go function is quadratic, it can be chosen:

$$J^*(x) = x^T S x, \quad S = S^T \succeq 0 \tag{2.18}$$

Then the gradient of this function is:

$$\frac{\partial J^*}{\partial x} = 2x^T S \tag{2.19}$$

Since all the terms are quadratic and convex by construction, we can find the solution of 2.17 by doing the gradient:

$$\frac{\partial}{\partial u} = 2u^T R + 2x^T S B = 0 \tag{2.20}$$

Then, the optimal policy is:

$$u^* = \pi^*(x) = -R^{-1} B^T S x = -Kx \tag{2.21}$$

By plugging 2.21 into 2.17:

$$0 = x^T[Q - SBR^{-1}B^TS + 2SA]x \tag{2.22}$$

where all the terms are symmetric except for 2SA, but since X^TSAx=x^TA^TSx:

$$0 = x^T[Q - SBR^{-1}B^TS + SA + A^TS]x \tag{2.23}$$

and since it has to hold for all x,

$$0 = Q - SBR^{-1}B^TS + SA + A^TS \tag{2.24}$$

that is a version of the **algebraic Riccati equation**.
Examining the optimal policy in more detail reveals interesting insights. The value function represents the cost-to-go, and the objective is to descend this landscape as quickly as possible. The direction of steepest descent of the value function is given by the negative of the state variable multiplied by the matrix S, i.e., -Sx. However, not all directions in the state-space are feasible to achieve.
To determine the steepest descent direction in the control space, we project the steepest descent onto the control space using the matrix B^T. This projection, $-B^TSx$, represents the steepest descent achievable with the control inputs u.
To account for different weightings placed on the control inputs, the pre-scaling by the matrix R^{-1} biases the direction of descent. This scaling ensures that the control inputs are appropriately weighted in the descent process.
By considering these factors, the optimal policy is designed to move in the direction of steepest descent while accounting for the limitations of the state-space and the weighting of control inputs.

Nonlinear systems

LQR can be used also with system that presents a nonlinear dynamics, because it can provide a local approximation of the optimal control solution.
Given a nonlinear system $\dot{x} = f(x, u)$ and a stabilizable operating point (x_0, u_0), with $f(x_0, u_0) = 0$, we can define:

$$\bar{x} = x - x_0, \quad \bar{u} = u - u_0 \tag{2.25}$$

and observe that $\dot{\bar{x}} = \dot{x} = f(x, u)$ Then we can approximate with a first-order Taylor expansion:

$$\dot{\bar{x}} \approx f(x_0, u_0) + \frac{\partial f(x_0, u_0)}{\partial x}(x - x_0) + \frac{\partial f(x_0, u_0)}{\partial u}(u - u_0) = A\bar{x} + B\bar{u} \tag{2.26}$$

Similarly, the cost function can be defined in the error coordinates or with a positive-definite second-order approximation of a nonlinear cost function around the operating point.
Therefore, optimal control solution is $\bar{u}^* = -K\bar{x}$ or $u^* = u_0 - K(x - x_0)$

Discrete formulation and finite horizon

The results above can be extended to discrete-time systems of the form:

$$x_{n+1} = Ax_n + Bu_n \tag{2.27}$$

where the problem is formulated as follows:

$$\min \sum_{n=0}^{N-1} [x_n^T Q x_n + u_n^T R u_n] \tag{2.28}$$

The cost-to-go is:

$$J(x, n-1) = \min_u x^T Q x + u^T R u + J(Ax + Bu, n) \tag{2.29}$$

If, as explained before, we take:

$$J(x, n) = x^T S_n x \tag{2.30}$$

then, the optimal control solution is

$$u_n^* = -K_n x_n = -(R + B^T S_n B)^{-1} B^T S_n A x_n \tag{2.31}$$

with

$$S_{n-1} = Q + A^T S_n A - (A^T S_n B)(R + B^T S_n B)^{-1}(B^T S_n A) \tag{2.32}$$

that is the **Riccati difference equation**.
In the finite-horizon formulation we have an extra term for the cost function:

$$J = h(x(t_f)) + \int_0^{t_f} \ell(x(t), u(t)) \tag{2.33}$$

for the continuous case and

$$J(x, u) = x_N^T Q_f x_N + \sum_{n=0}^{N-1} [x_n^T Q x_n + u_n^T R u_n] \tag{2.34}$$

for the discrete case.
The first term is the final cost term and it only depends on the state of the system.

Chapter 3

Methodology

In the previous chapter, the different possibilities of dealing with manipulation and control of deformable objects were described, as well as the reasons that brought to the choice of a *model-based* approach to solve the proposed problem. In Section 2.1.1, it was highlighted that there are two major types of methods for this kind of solution: *direct* or *indirect* methods.
It is needed to recall that the problem has the form shown in 2.1, where inequality constraints and system's dynamics are present, making it a **constrained optimisation problem**. This lead to the choice of the implementation of a **indirect** method for its solution. Indeed, it was proven that this approach result in a faster computation since it enforce the dynamics implicitly by simulating the system's dynamics, whilst with direct methods it has to be imposed slowing down the process.
Among these, DDP was proven to be one of the most successful along with the fact that it can be modified to include other type of constraints, as it does not handle them in the original form. For sake of simplicity however, iLQR will be implemented because of the lower required computational cost and the fact that it was proven to be faster in time and iterations.
To deal with the constraints, there are several approaches: among the different possibilities, the Augmented Lagrangian and Multipliers Method (ALMM) was chosen because of the intuitiveness and simplicity of the method itself as it has been also proven as efficient in literature.
Therefore, the implemented solver will be the following one:

Algorithm 2 AL-iLQR

```
function AL-iLQR(x_0,U,tol.)
    ▷ Initialize λ, μ, φ
    while max(c)>tol. do
        ▷ minimize L_A(X,U,λ,μ) using iLQR
        ▷ update the multipliers λ and the penalty term μ
    end while
return X,U,λ
end function
```

For what concerns the model for the non-rigid object, accurate models, like those associated with FEM, are more appropriate for off-lines simulation; here, however, the choice of manipulating a DLO, like a rope, leads to a *mass-spring-damped model* for modeling it.
In general, the choice of the suitable model relies on the available programming time and computational resources and the desired accuracy. Physical analogues like mass-spring models are commonly used due to their ease of implementation and tuning, they also provide a good trade-off between speed and accuracy, that can be adjusted by altering the number of nodes and springs.
In the next Section, the above algorithm will be described in details, as well the model of the rope and the chosen cost function.

3.1 Structure of the Algorithm

Algorithm 2 highlights the two main components: the **Augmented Lagrangian Multiplier Method** and the **iterative-Linear-Quadratic-Regulator**. Here, the theory behind both of them is explained recalling what has been said in the previous chapter.

3.1.1 Augmented Lagrangian and Multipliers Method

The ALMM relies on the idea of *relaxation* and *penalty functions*.
The first consists of techniques that remove constraints from the problem to allow for a version, i.e. a relaxation, that is simpler to solve and/or provide information to be used for solving the original problem.
The second instead relies on the idea of converting the constrained optimization problem into an unconstrained one that is augmented with a penalty function, that penalises violations of the original constraints.

In general, the constrained problem P has the following form:

$$\begin{aligned}(P) : \min\ & f(x) \\ \text{s.t. } & g(x) \leq 0 \\ & h(x) = 0 \\ & x \in X\end{aligned} \tag{3.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, $h : \mathbb{R}^n \to \mathbb{R}^l$, and $X \subseteq \mathbb{R}^n$ is an open set.
Then, when one refers to *relaxation*, for a given set of *dual variables (u,v)* $\in \mathbb{R}^{m+l}$ with $u \geq 0$, the original problem P is transformed into:

$$(D) \ : \ \theta(u, v) = \inf_{x \in X} \phi(x, u, v) \tag{3.2}$$

where

$$\phi(x, u, v) := f(x) + u^T g(x) + v^T h(x) \tag{3.3}$$

is called the **Lagrangian function**, the original problem is called *primal* problem, the relaxation is called *(Lagrangian) dual* problem and $\theta(u, v)$ is the Lagrangian dual function, which has a built-in optimization problem in x, meaning that evaluating $\theta(u, v)$ requires finding the minimiser $\bar{x}$ for $\phi(x, u, v)$, given (u, v).
Whereas, the **penalised version** of (P) in 3.1 is:

$$(P_\mu) : \min .\{f(x) + \mu\alpha(x) : x \in X\} \tag{3.4}$$

where $\mu > 0$ is a *penalty term* and $\alpha(x) : \mathbb{R}^n \to \mathbb{R}$ is a *penalty function* of the form

$$\alpha(x) = \sum_{i=1}^{m} \phi(g_i(x)) + \sum_{i=1}^{l} \psi(h_i(x)) \tag{3.5}$$

For $\alpha(x)$ to be a suitable penalty function, must hold that $\phi : \mathbb{R}^n \to \mathbb{R}$ and $\psi : \mathbb{R}^n \to \mathbb{R}$ are continuous and satisfy

$$\begin{aligned}\phi(y) = 0 \ if \ y \leq 0 \text{ and } \phi(y) > 0 \ if \ y > 0 \\ \psi(y) = 0 \ if \ y = 0 \text{ and} d \ \psi(y) > 0 \ if \ y \neq 0\end{aligned} \tag{3.6}$$

In this way, the method will solve constrained problems by moving the constraints into the cost function and iteratively increasing the penalty for either getting closer or violating the constraint. However, this method converges only if the penalty terms go to infinite, which is impractical.
A way to seek for the exact convergence with a finite penalty term is the implementation of the **ALMM**, that maintains the estimates of the Lagrange multipliers associated with the constraints. [29]

The term *augmented* refers to the fact that the Lagrangian function in 3.3 is augmented by a penalty term as shown below:

$$\mathcal{L}_A = f(x) + \lambda^T c(x) + \frac{1}{2} c(x)^T I_\mu c(x) \tag{3.7}$$

where λ are the Lagrange multipliers, $c(x)$ is a vector containing both inequality and equality constraints and I_μ is a diagonal matrix where the diagonal elements are the penalty multipliers μ_i and it is computed as:

$$I_\mu = \begin{cases} 0 & \text{if } c_i(x) < 0 \wedge \lambda_i = 0, i \in \mathcal{I} \\ \mu_i & \text{otherwise} \end{cases} \tag{3.8}$$

This allows for noticing that if $(\bar{x}, \bar{\lambda})$ is a solution of the problem, then the optimality condition:

$$\nabla_x \mathcal{L}_A = \nabla f(x) + \sum_{i=1}^{l+m} \bar{\lambda} \nabla c_i(x) + 2 I_\mu \sum_{i=1}^{l+m} c_i(x) \nabla c_i(x) = 0 \tag{3.9}$$

implies that the optimal solution can be achieved with a finite penalty term.
Then, the discrete-time problem in 2.1 is transformed into:

$$\begin{aligned} &\mathcal{L}_A(X, U, \lambda, \mu) = \ell_N(x_N) + (\lambda_N + \frac{1}{2} I_{\mu_N} c_N(x_N))^T c_N(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k, \Delta t) + \\ &(\lambda_k + \frac{1}{2} I_{\mu_k} c_k(x_k, u_k))^T c_k(x_k, u_k) \\ &= \mathcal{L}_N(x_N, \lambda_N, \mu_N) + \sum_{k=0}^{N-1} \mathcal{L}_k(x_k, u_k, \lambda_k, \mu_k) \end{aligned} \tag{3.10}$$

where $\lambda_k \in \mathbf{R}^{p_k}$ is a Lagrange multiplier, c is the vector containing the constraints, $\mu_k \in \mathbf{R}^{p_k}$ is the penalty terms associated to inequality constraints with index sets $\mathcal{I}_k$ and equality constraints with index set $\mathcal{E}_k$. Then, I_{μ_k} is a diagonal matrix defined as,

$$I_{\mu_k, ii} = \begin{cases} 0 & \text{if } c_{k_i}(x_k, u_k) < 0 \wedge \lambda_{k_i} = 0, \ i \in \mathcal{I} \\ \mu_{k_i} & \text{otherwise} \end{cases} \tag{3.11}$$

where k_i indicates the *i-th* constraint at time step k.
In this way, it is possible to employ an unconstrained optimisation method to solve the augmented Lagrangian function as showed in Algorithm 2:

1. solve $\min_x \mathcal{L}_A(x, \lambda, \mu)$, holding λ and μ constant;

2. update Lagrange multipliers

$$\lambda_i^+ = \begin{cases} \lambda_i + \mu_i c_i(x^*) & i \in \mathcal{E} \\ \max(0, \lambda_i + \mu_i c_i(x^*)) & i \in \mathcal{I} \end{cases} \tag{3.12}$$

   this because the correspondent multiplier is 0 when the inequality constraint is not active;

3. update penalty term: $\mu^+ = \phi\mu,\ \phi > 1$;

4. check constraint convergence;

5. if tolerance not met, go to 1.

3.1.2 Differential Dynamic Programming and iterative Linear Quadratic Regulators

To solve the augmented lagrangian showed in 3.10, the iLQR method is employed, that is a simplified version of DDP.
DDP refers to a general class of dynamic programming algorithms that iteratively solve finite-horizon discrete-time control problems by using locally quadratic models of cost and dynamics, as explained in Section 2.3.1.
Considering a discrete-time dynamics model for state, control pair (x,u) with a dynamic as $x_{t+1} = f(x_t, u_t)$, let u be a control sequence $u_0, u_1, ..., u_n$, the total cost J is defined as the sum of the cost-to-go at each time step and the terminal cost as in 2.34. The solution is a control sequence that minimizes the total cost

$$u^* = argmin_u J(x, u) \tag{3.13}$$

With respect to the LQR algorithm in Section 2.3.1, DDP doesn't require a quadratic cost and a linear dynamics; it is more general and powerful as a general framework for solving optimal control problems. In fact, it makes use of a second-order quadratic approximation of both cost function and dynamics and iteratively solves the problem to find the optimal control sequence backwards in time recursively.
In addition to what has been said in section 2.3.1, in DDP the aim is to optimize with respect to a nominal trajectory

$$\hat{\tau} = \{(\hat{x_0}, \hat{u_0}), (\hat{x_1}, \hat{u_1}), .., (\hat{x_{n-1}}, \hat{u_{n-1}})\} \tag{3.14}$$

Defined (x_i^*, u_i^*) as to be the optimal state,control pair for time step i, then we can define the optimal perturbations $(\delta x_i^*, \delta u_i^*)$

$$\begin{aligned} x_i^* &= \hat{x_i} + \delta x_i^* \\ u_i^* &= \hat{u_i} + \delta u_i^* \end{aligned} \tag{3.15}$$

Then, it is convenient to recast the problem of finding the optimal control sequence to a separate and equivalent problem of finding the optimal control perturbation on some nominal trajectory as:

$$\begin{aligned} V(x,i) &= \min_{u}[l(x,u) + V(f(x,u), i+1)] \\ V(x,i) &= \min_{\delta u}[l(\hat{x}+\delta x, \hat{u}+\delta u) + V(f(\hat{x}+\delta x, \hat{u}+\delta u), i+1)] \end{aligned} \tag{3.16}$$

Now, the problem is formulated with respect to the perturbation from a nominal trajectory, that's why it is called *differential*.
In general, the DDP algorithm can be decomposed in 3 steps: backward-pass, forward-pass and line search.
To understand the difference of the iterative Linear Quadratic Regulator with the DDP, one needs to look into the Q-function, defined as:

$$Q(\delta x, \delta u) = \Big(\ell(\hat{x}+\delta x, \hat{u}+\delta u) - \ell(\hat{x},\hat{u})\Big) + \Big(V(f(\hat{x}+\delta x, \hat{u}+\delta u)) - V(f(\hat{x},\hat{u}))\Big) \tag{3.17}$$

The Q-function is a scalar function taking vector inputs and it expresses the change in cost that results from perturbing a point in the nominal trajectory $\hat{\tau}$. The goal is to find perturbations that minimize the Q-function.
Below there are expressed the derivatives of this function:

$$\begin{aligned} Q_x &= \frac{\partial Q}{\partial x} \\ Q_u &= \frac{\partial Q}{\partial u} \\ Q_{xx} &= \frac{\partial}{\partial x}\frac{\partial}{\partial x}Q \\ Q_{uu} &= \frac{\partial}{\partial u}\frac{\partial}{\partial u}Q \\ Q_{xu} &= \frac{\partial}{\partial x}\frac{\partial}{\partial u}Q \end{aligned} \tag{3.18}$$

The second-order approximation of the Q-function can be then written as:

$$Q(\delta x, \delta u) \approx \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} 0 & Q_x^T & Q_u^T \\ Q_x & Q_{xx} & Q_{xu} \\ Q_u & Q_{ux} & Q_{uu} \end{bmatrix} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix} \tag{3.19}$$

Where

$$\begin{aligned} Q_x &= l_x + f_x^T V_x' \\ Q_u &= l_x + f_u^T V_x' \\ Q_{xx} &= l_{xx} + f_x^T V_{xx}' f_x + (V_x' f_{xx}) \\ Q_{uu} &= l_{uu} + f_u^T V_{xx}' f_u + (V_x' f_{uu}) \\ Q_{ux} &= Q_{xu}^T = l_{ux} + f_u^T V_{xx}' f_x + (V_x' f_{ux}) \end{aligned} \tag{3.20}$$

The terms in the round brackets describe the difference between DDP and iLQR: the latter in fact uses a linear approximation instead of a quadratic one. This results in the cancellation of a few terms in the 2^{nd} order expansion coefficient of the Q-function derivatives.
The algorithm is shown below:

Algorithm 3 iterative LQR

1: **function** iLQR(x_0,U,tol.)
2: ▷ Initialize x_0,U,tol.
3: **while** $|J - J^-| >$ tol. **do**
4: $J^- \leftarrow J$
5: $K, d, \Delta V \leftarrow$ BACKWARDPASS(x, u)
6: $x, u, J \leftarrow$ FORWARDPASS$(x, u, K, d, \Delta V, J^-)$
7: **end while**
8: **return** x,u,J
9: **end function**

It has two main steps: **backward pass** and **forward pass**, which are shown in the pseudo-algorithm 4 and 5 respectively.
In the backward pass, there is the computation of the matrices needed for computing the optimal control input u.
Here, to handle the constraints, some changes have to be made to the LQR previously explained in Section 2.3.1. The main change concerns the equation 3.16, where the cost-to-go has to be replaced by the Lagrangian 3.10, that leads to the new Q-function derivatives:

$$\begin{aligned} Q_x &= l_x + A^T p' + c_x^T(\lambda + I_\mu c) \\ Q_u &= l_x + B^T p' + c_u^T(\lambda + I_\mu c) \\ Q_{xx} &= l_{xx} + A^T P' A + c_x^T I_\mu c_x \\ Q_{uu} &= l_{uu} + B^T P' B + c_u^T I_\mu c_u \\ Q_{ux} &= Q_{xu}^T = l_{ux} + B^T P' A + c_u^T I_\mu c_x \end{aligned} \tag{3.21}$$

where $A = \frac{\partial f}{\partial x}|_{x_k,u_k}$, $B = \frac{\partial f}{\partial u}|_{x_k,u_k}$ and ' indicates variables at time step k+1.
By optimising 3.19 with respect to the correction to the control trajectory, the following result is achieved:

$$\delta u_k^* = -(Q_{uu} + \rho I)^{-1}(Q_{ux}\delta x_k + Q_u) = K_k \delta x_k + d_k \tag{3.22}$$

where the regularization term is added to handle poor conditioned Hessians.
By substituting into 3.19, we derive:

$$\begin{aligned} P_k &= Q_{xx} + K_k^T Q_{uu} K_k + K_k^T Q_{ux} + Q_{xu} K_k \\ p_k &= Q_x + K_k^T Q_{uu} d_k + K_k^T Q_u + Q_{xu} d_k \end{aligned} \tag{3.23}$$

$$\Delta V_k = d_k^T Q_u + \frac{1}{2} d_k^T Q_{uu} d_k \tag{3.24}$$

For $k = N$, at the terminal time-step, there are no control to optimize, hence p_N and P_N are computed in the following way:

$$\begin{aligned} P_N &= (\ell_N)_{xx} + (c_N)_x^T I_{\mu_N} (c_N)_x \\ p_k &= (\ell_N)_x + (c_N)_x^T (\lambda + I_{\mu_N} (c_N)_x) \end{aligned} \tag{3.25}$$

The algorithm is shown below:

Algorithm 4 Backward Pass

```
1: function BACKWARDPASS(x, u)
2:     p_N, P_N ← (3.25)
3:     for k=N-1:-1:0 do
4:         δQ ← (3.21)
5:         if Q_uu ≻ 0 then
6:             K, d, ΔV ← (3.22), (3.24)
7:         else
8:             Increase ρ and go to line 3
9:         end if
10:    end for
11: return K, d, ΔV
12: end function
```

Once the optimal feedback gains for each time step are computed, the nominal trajectories can be updated in the *forward pass* by simulating forward the dynamics:

$$\begin{aligned} \delta x_k &= \bar{x}_k - x_k \\ \delta u_k &= K_k \delta x_k + \alpha d_k \\ \bar{u}_k &= u_k + \delta u_k \\ \bar{x}_{k+1} &= f(\bar{x}_k, \bar{u}_k) \end{aligned} \tag{3.26}$$

where $\bar{x}_k$ and $\bar{u}_k$ are the updated nominal trajectories and $0 \leq \alpha \leq 1$ is a scaling term.
The algorithm is shown below:

Algorithm 5 Forward Pass

1: **function** FORWARDPASS($X, U, K, d, \Delta V, J$)
2: Initialize $\bar{x_0} = x_0, \alpha = 1, J^- \leftarrow J$
3: **for** k=0:1:N-1 **do**
4: $\bar{u_k} = u_k + K_k(\bar{x_k} - x_k) + \alpha d_k$
5: $x_{k+1} \leftarrow$ Using $\bar{x_k}, \bar{u_k}$
6: **end for**
7: $J \leftarrow$ Using X,U
8: **if** J satisfies line search conditions **then**
9: $X \leftarrow \bar{X}, U \leftarrow \bar{U}$
10: **else**
11: Reduce α and go to line 3
12: **end if**
13: **return** X, U, J
14: **end function**

Since the optimization problem is nonlinear, a line search along the descent direction is needed to ensure an adequate reduction in cost. The used one is a simple *backtracking* line search on the feed-forward term using the parameter α as explained previously in Section 2.2.

3.2 Model of the Rope: Damped-Spring-Mass system

The model of the rope is designed through point masses linked by springs. Each point is characterized by position, velocity and mass even though it doesn't take space. In this way, the state vector of the rope will be a vector of length 6 by the number of the masses:

$$x = \begin{bmatrix} p \\ v \end{bmatrix}$$

where p stores the positions of all the masses in a 3D space and v the corresponding velocities.
The adopted model is shown below:

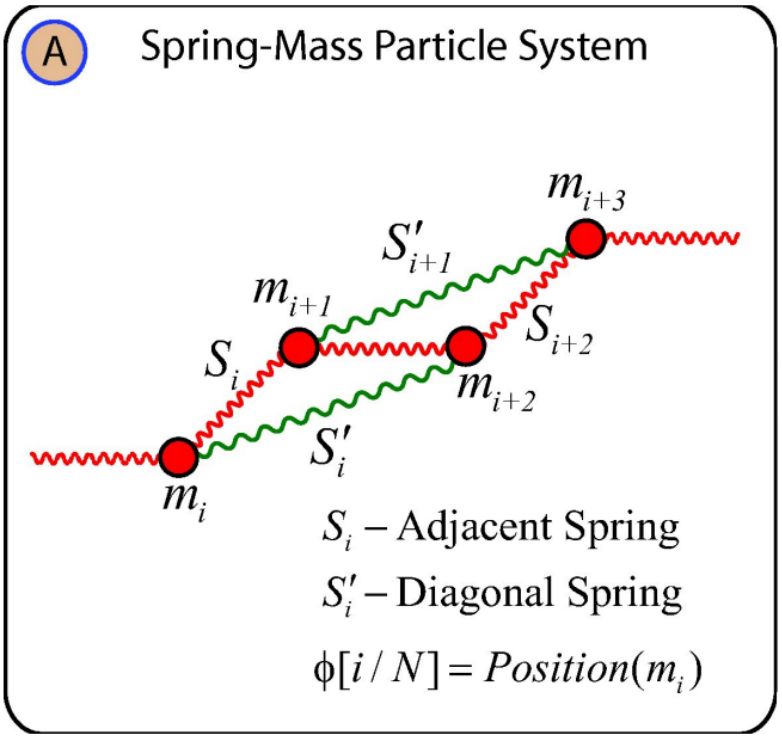

Figure 3.1: Mass-Spring-Damped Model [32]

Initially, the object is in a rest shape denoted by $\mathbf{S}_0$. When an external force $\mathbf{f}_{ext}$, such as gravity or force applied by a manipulator, is exerted on the object, it undergoes deformation, causing the masses within the object to move to a new position represented by $\mathbf{p}^{new}$.
In physics-based models, the resulting deformation is typically quantified using the displacement vector $\Delta\mathbf{p}$, which is calculated as the difference between $\mathbf{p}^{new}$ and the initial position $\mathbf{p}^0$. Additionally, the displacement unit vector $\mathbf{u}$ is defined as the normalized form of the displacement, obtained as $\frac{\Delta\mathbf{p}}{||\Delta\mathbf{p}||}$.
To model the dynamic behavior of deformations over time, Newton's second law of motion is applied. This law relates the forces acting on the object to its resulting acceleration, enabling the simulation of the object's dynamic response to external forces.
Let $\mathbf{p}_i^t$ be the position of the particle i at the time t:

$$\mathbf{v}_i^t = \dot{\mathbf{p}}_i^t, \quad \mathbf{a}_i^t = \dot{\mathbf{v}}_i^t, \quad m_i\mathbf{a}_i^t = \mathbf{f}_{ext_i}^t \tag{3.27}$$

where m_i, $\mathbf{f}_{ext_i}^t$, $\mathbf{a}_i^t$ and $\mathbf{v}_i^t$ are respectively the mass, the external forces, acceleration and velocity at time t and $\dot{\mathbf{p}}_i^t$, $\dot{\mathbf{v}}_i^t$ are first-order time derivatives of the position and velocity [33]. The Hooke's law is used to simulate the springs, which states that the force exerted by a spring scales linearly with how much it is elongated or

compressed:

$$\mathbf{F} = -k(\Delta\mathbf{p} - l_{rest}) \tag{3.28}$$

where $\mathbf{k}$ is the spring stiffness and it is constant, $\mathbf{u}$ is the unit vector along the spring computed earlier, i.e. the normalized version of the distance between the two endpoints of the springs, $\Delta\mathbf{p}$ is the current length of the spring and l_{rest} is the length of the spring at rest.
The Hooke's law gives the force exerted by the spring on the masses once known their positions.
When dealing with dynamical manipulation of the object, there are unrealistic behaviours if the damping factor is not introduced. Hence, in addition to the Hooke's law, there is also a damping factor $\mathbf{F}_{damped} = -c\Delta\dot{\mathbf{p}}$ along with the forces derived from the gravitational acceleration $\mathbf{g}$ that goes along the negative z direction. Thus, the total force is given by:

$$\mathbf{F} = -k(\Delta\mathbf{p} - l_{rest}) - c\Delta\dot{\mathbf{p}} + m\mathbf{g} \tag{3.29}$$

Moreover, there is the external force given by the control input that has to be added to one of the masses at the end of the rope in order to manipulate it. Hence: for the manipulated mass, the total force is:

$$\mathbf{F} = -k(\Delta\mathbf{p} - l_{rest}) - c\Delta\dot{\mathbf{p}} + m\mathbf{g} + F_{input} \tag{3.30}$$

To ensure stability and simulate shear/strain forces in the original shape formation, it is necessary to incorporate three types of springs: structural (elastic) springs, shear springs, and bend (flexion) springs.
The structural springs help maintain the elasticity of the model and provide resistance against deformation. They contribute to the restoration of the object to its original shape when external forces are applied.
Shear springs are essential for preventing excessive stretching or elongation of the model in diagonal directions. They restrict the deformation of the object and maintain its stability.
Bend springs, on the other hand, exert forces to counteract any bending or folding of the model. They assist in preserving the structural integrity of the object and ensure that it retains its desired shape.
Referring to Figure 3.1, the adjacent springs are the elastic and shear ones, whilst the diagonal are the flexion ones. Thus, the internal forces that act on two adjacent masses are:

$$\mathbf{F} = -k_{elastic}(\Delta\mathbf{p} - l_{rest}) - c_{elastic}\Delta\dot{\mathbf{p}} - k_{shear}(\Delta\mathbf{p} - l_{rest}) - c_{shear}\Delta\dot{\mathbf{p}} \tag{3.31}$$

whereas for two diagonal masses holds:

$$\mathbf{F} = -k_{bend}(\Delta\mathbf{p} - l_{rest}) - c_{bend}\Delta\dot{\mathbf{p}} \tag{3.32}$$

This means that on each mass, the internal force acting on it is the linear combination of the elastic and shear force from the adjacent masses and the bend force from the diagonal ones.
The inclusion of additional types of springs, such as shear and bend springs, in addition to the structural springs, may not be immediately apparent. This arises from the inherent one-dimensional nature of conventional spring models. This limitation, indeed, necessitates the introduction of shear and bend springs to capture more complex deformations accurately. Shear springs help prevent excessive stretching or elongation of the object in diagonal directions, while bend springs resist bending or folding of the model.
Consequently, constructing an appropriate spring network often relies on *prior knowledge* or an iterative process of *trial and error* to ensure stability and capture the desired deformations of the specific object. [34]
Here, the physical parameters of the rope are set based on a trial-and-error procedure as a trade-off between computational issues and the ones that represent in a better way the behavior of a rope. They are resumed in the table below.

Parameter	Value
N° of masses	5
Rest length of the springs	1
Total mass	0.3
Stiffness of elastic springs	140
Stiffness of shear springs	5
Stiffness of bend springs	1
Damping value of elastic springs	28
Damping value of shear springs	1
Damping value of bend springs	0.2

Table 3.1: Parameters of the rope [35]

Once the parameters are defined, to advance in time we can iteratively evaluate the current rate of change and take small steps forward by using an integration scheme as:

$$x_{k+1} = x_k + change\_of(x_k)dt \tag{3.33}$$

In order to use the algorithm previously described, we need the Jacobians of the dynamics with respect to the state and the control input. To get them, it is possible to use JAX environment in Python.
JAX is a powerful tool that extends the capabilities of NumPy to run on various hardware platforms such as CPU, GPU, and TPU. It provides excellent support for

automatic differentiation, making it ideal for high-performance machine learning research.
One of the key features of JAX is its ability to automatically differentiate native Python and NumPy code. It supports a wide range of Python features, including loops, conditional statements (ifs), recursion, and closures. This means that JAX can handle complex code structures and calculate derivatives of functions that involve these features.
Additionally, JAX offers the ability to compute higher-order derivatives, allowing for calculations of derivatives of derivatives of derivatives and so on.
Here, since it is needed to differentiate a vector-valued function, i.e. the dynamics, that produces the Jacobian, the most suitable function to use is **jax.jacfwd** due to the dimensionality of the matrix nxm where n is greater than m.

3.3 The cost function

The other element that contributes to the efficiency of the optimization process is the employment of a suitable cost function.
In optimal control, the cost function plays a crucial role in determining the optimal control policy. The cost function represents the objective that the control policy seeks to optimize over a specified time horizon. It is a mathematical expression that quantifies the performance of the system being controlled based on the control inputs and the states of the system.
The role of the cost function is to provide a quantitative measure of how well the system is performing under a given control policy. The optimal control problem aims to find the control policy that minimizes the cost function while satisfying system constraints. The choice of cost function depends on the specific application and the performance criteria of the system being controlled.
The cost function typically includes two components: a state cost and a control cost. The state cost penalizes the deviation of the system state from a desired set-point, while the control cost penalizes the magnitude of the control inputs. The relative weights assigned to these components determine the trade-off between the performance of the system and the cost of control.
For the describe problem the cost function of the LQR is expressed as:

$$J(x,u) = (x_N - x_{goal})^T Q_f (x_N - x_{goal}) + \sum_{k=0}^{N-1} [(x_k - x_{goal})^T Q (x_k - x_{goal}) + u_k^T R u_k] \tag{3.34}$$

where Q, Q_f and R are diagonal and positive definite matrices.
The x vector contains the position and the velocities of all the masses of the rope, so it aims to minimize the distance between the current position and the target position. The goal is to manipulate the rope in a such a way that it lies on the box

as shown in Figure 1.2.
The cost function has to include the goal state because the LQR controller, as it is constructed, goes to the zero state by default. In this way, a reference is set and the minimization is done in order to achieve a specific position.
For what concerns the starting position, it was chosen to represent the resting position of the robotic arm where the end-effector holds one of the ends of the rope. In this way the first mass is in position (0,0,4) and the last one lays on the ground.
The cost function can be chosen in multiple ways. The goal state, as implemented in the code, has the manipulated mass in (1,0,4.2) and the last mass in (5,0,4.2). The weight of the final position has the highest influence on the cost function. However, the most important aspect is that the end of the rope lays on the box, which means that the manipulated mass is not forced to reach the goal state (1,0,4.2). In this sense, a heavier weight on the last masses is a more efficient way to proceed.
In Figure 3.2 and Figure 3.3, it is shown the trajectory and the final position of the manipulated rope with respectively an equal weight for all the masses and a weight more significant for the last 3 ones. The weight is set to 150 for high influence and 10^{-4} for low influence.
Here, the velocity of the last time-step is neglected, hence the weight is set to 150 only for the positions in the state vector, whilst for velocities is set to 10^{-4}.

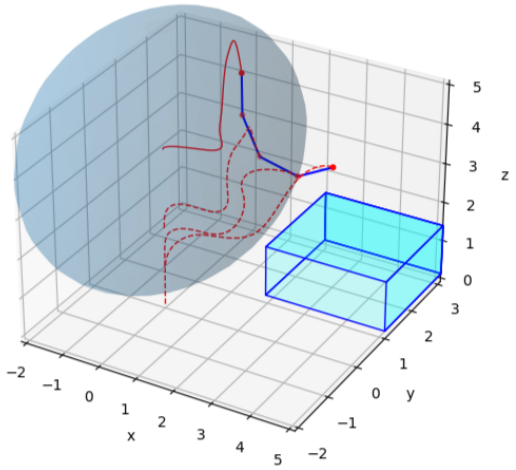

Figure 3.2: Trajectory and final position with a cost function that penalizes the last three masses

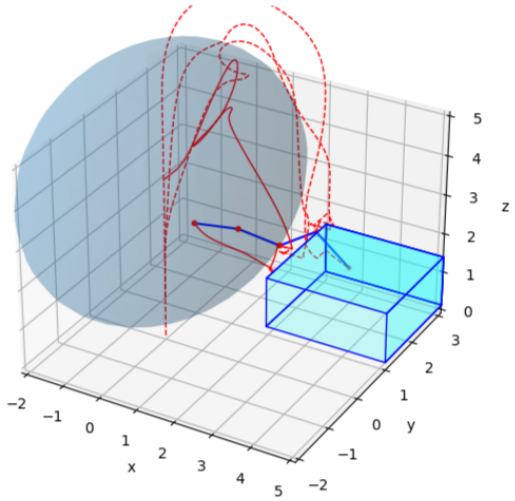

Figure 3.3: Trajectory and final position with a cost function that penalizes all the masses equally

Both figures show the trajectories of the manipulated mass and the masses involved in the cost function and the final position in output after 1 iteration of the

algorithm, but this is enough to show that the one that penalizes only the relevant masses has a smoother trajectory and reaches the goal in a better way, whereas the other one has a worse behaviour since it tries also to get the manipulated mass closer to the target position, that is unnecessary for this task.

3.4 Discretization of the model and the role of the time-step

Once the model of the rope with its equations is defined, one should notice the nonlinearity of it. Indeed, the dynamical model of the rope is a *nonlinear* and *continuous* model of the form:

$$\dot{x}(t) = f(x(t), u(t)) \tag{3.35}$$

By discretizing it, the controller can be applied, i.e. the continuous-time nonlinear model has to be approximated by a discrete-time nonlinear model.
The discretization process involves sampling the continuous-time model at discrete time intervals and approximating the continuous-time dynamics by a *difference equation* that describes the evolution of the state variables between samples. The choice of the sampling interval, or the time step, is critical in determining the accuracy and stability of the discrete-time model.
There are several methods for discretizing nonlinear models, including the *Euler method*, *semi-implicit methods* like backward Euler method, trapezoidal rule, and higher-order numerical integration methods such as *Runge-Kutta methods*. These methods involve approximating the nonlinear differential equations that describe the continuous-time dynamics by difference equations that describe the discrete-time dynamics.
Here, the main differences and advantages are described for each of them:

1. Euler's Method:
    - Euler's method is a simple and straightforward approach to numerical integration.
    - It uses a first-order approximation to estimate the solution at the next time step.
    - It is explicit, meaning that the solution at the next time step is solely based on the current solution.
    - Euler's method has a local truncation error of $O(h^2)$, where h is the time step size.
    - It is computationally efficient but less accurate compared to higher-order methods.

- Euler's method can exhibit stability issues, especially for stiff systems or large time step sizes.

2. Semi-Implicit Methods:

   - Semi-implicit methods combine implicit and explicit components in their formulation.
   - These methods update some variables explicitly while updating others implicitly.
   - Implicit updates involve solving algebraic equations at each time step, making them more computationally intensive compared to explicit methods.
   - By incorporating implicit updates, semi-implicit methods can provide better stability and accuracy compared to explicit methods like Euler's method.
   - Common semi-implicit methods include the backward Euler method, the trapezoidal rule, and the midpoint rule.

3. Runge-Kutta 4th order Methods:

   - The fourth-order Runge-Kutta method is a widely used numerical integration technique.
   - It is an explicit method that computes the solution at the next time step based on weighted averages of function evaluations at different points within the time interval.
   - RK4 employs four function evaluations per time step, which improves accuracy compared to Euler's method.
   - It has a local truncation error of $O(h^5)$, making it more accurate than Euler's method and semi-implicit methods.
   - RK4 is more stable than Euler's method for a wide range of problems and can handle stiff systems with appropriate time step sizes.
   - However, RK4 is computationally more expensive than Euler's method and semi-implicit methods due to the additional function evaluations.

In summary, Euler's method is simple but less accurate and prone to stability issues. Semi-implicit methods strike a balance between explicit and implicit approaches, providing improved stability and accuracy. RK4 is a higher-order explicit method that offers better accuracy at the cost of increased computational complexity.
In the following tables the equations are shown where the terms t and y are referred to time-step n, while $'$ refers to time-step $n+1$.

Euler method	Semi-Implicit	RK4
$y' = y + hf(t, y)$	$y' = y + hf(t', y')$	$k_1 = f(t, y),$ $k_2 = f(t + \frac{h}{2}, y + h\frac{k_1}{2})$ $k_3 = f(t + \frac{h}{2}, y + h\frac{k_2}{2})$ $k_4 = f(t + h, y + hk_3)$ $y' = y + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h$

Table 3.2: Systems equation of the discretization methods described above

It is important to note that discretization of nonlinear models introduces *discretization errors*, which can lead to discrepancies between the simulated or controlled system and the actual system. Therefore, it is important to carefully evaluate the discretization method and the choice of the time step to ensure that the discretization errors are small and do not significantly affect the performance or stability of the system.
The **time-step** determines the *interval* at which the continuous-time system is sampled to obtain a discrete-time representation.The choice of time-step size has significant implications for the accuracy, stability, and computational efficiency of the resulting discrete-time model.
A *smaller time-step* size generally leads to a *more accurate discretization*. When the time-step is small, the discrete-time model better approximates the behavior of the original continuous-time system. However, using a very small time-step can *increase computational requirements* and may not always be necessary depending on the dynamics of the system.
The time-step size affects the *stability* of the discrete-time system. It can introduce stability issues if the time-step is too large. In general, a smaller time-step size improves stability by preventing large errors or instability due to nonlinear behavior.
A larger time-step size reduces the number of calculations required to obtain the discrete-time model, thus improving computational efficiency. However, a larger time-step can result in a less accurate representation of the system's dynamics, potentially leading to performance degradation. Balancing computational efficiency and accuracy is crucial when selecting an appropriate time-step.
In general, the optimal time step is often determined by *trial and error*, starting with a relatively large time step and gradually decreasing it until the desired level of accuracy is achieved, or until the computational cost becomes prohibitive.
In Section 2.1.1, it has been emphasized that when dealing with high stiffness materials, mass-spring system techniques face stability issues, thereby necessitating a smaller time integration step in the simulation process. For this reason, even

though it's more computationally expensive, RK4 is needed to have a stable simulation.
Furthermore, it can be shown that, when the time-step is higher than 10^{-3}, the parameters of the rope in Table 3.1 have to be set to lower values otherwise they produce *Nan* values when the transition of the dynamics happens. By having a time-step of 10^{-3}, the computational cost is more expensive and it is needed a length horizon of 2000.
The horizon is an important parameter in optimal control and the performances of the algorithm against different lengths for it will be made in the next chapter.
All these considerations lead to the decision of setting two different time-steps, a smaller one for the integration time, such that it is possible to achieve a more accurate simulation of the dynamical behavior of the rope and a slightly bigger one for the controller. The analysis on the different rates of frequency between them is made in the next chapter since also the time-step of the controller is an important feature that describes the performances of the algorithm.
Here it is shown the result achieved with an integration time-step of 0.001 and a controller time-step of 0.01.

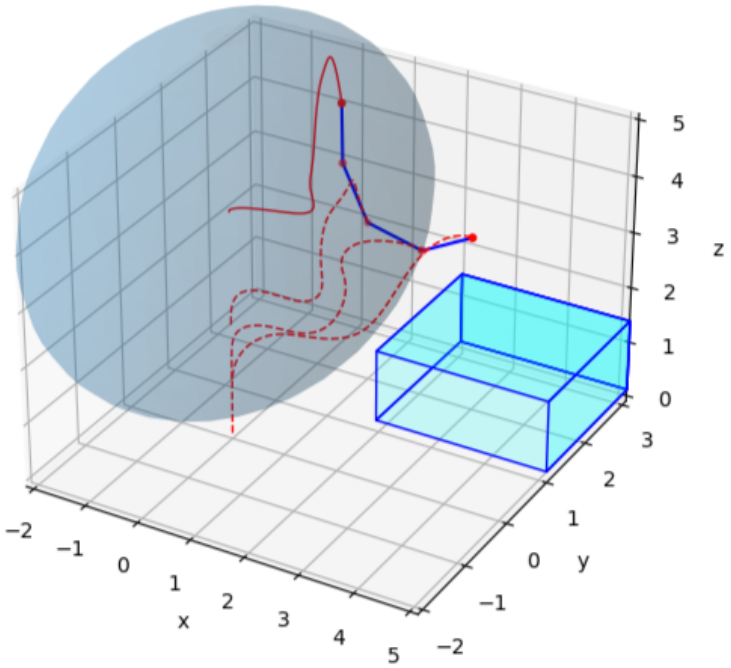

Figure 3.4: Trajectory and final position with time-step of 0.001 for the integration time and 0.01 for the controller

In order to simulate the rope dynamic of the form 3.35, the control input u is needed along all the simulation steps: for this reason, the algorithm computes the suitable control input every 10 steps of the simulation horizon and it is applied over the next 10 integration time-steps to roll-out the dynamic of the rope.
Thus, the forward pass of the LQR is modified as follows:

Algorithm 6 Modified Forward Pass

1: **function** MODIFIED-FORWARDPASS($X, U, K, d, \Delta V, J$)
2: Initialize $\bar{x_0} = x_0, \alpha = 1, J^- \leftarrow J$
3: **for** k=0:10:N-1 **do**
4: $\bar{u_k} = u_k + K_k(\bar{x_k} - x_k) + \alpha d_k$
5: **for** l=0:1:10 **do**
6: $x_{k+l+1} \leftarrow$ Using $\bar{x_{k+l}}, \bar{u_k}$
7: **end for**
8: **end for**
9: $J \leftarrow$ Using X,U
10: **if** J satisfies line search conditions **then**
11: $X \leftarrow \bar{X}, U \leftarrow \bar{U}$
12: **else**
13: Reduce α and go to line 3
14: **end if**
15: **return** X, U, J
16: **end function**

3.5 Structure of the code

The code was implemented in Python making use of *classes*.
The classes are:

- Rope, *rope_system.py*;
- LQR, *iLQR.py*;
- AL-iLQR, *AL_iLQR.py*;
- Constraints, *constraints.py*.

3.5.1 Rope

The model of the rope is defined in the python file *rope_system.py* and shown in the appendix section A.1.
It is implemented as a class, inherited from another one called *DynamicalSystem*

which defines the cost function and the matrices Q and R related to the LQR, described previously. Its methods compute also the running cost and the final one and set the target.
Then, the Rope class contains all the needed methods to set the initial and final positions, the goal state and the dynamic of the system.
Here the hooked-damped law explained earlier is implemented; the internal forces and the gravity one are then applied to all the masses, whilst the external force, the control law, is applied only to the actuated one.
The derivative of the state vector, i.e. velocity and acceleration, is computed in the function *change_of_state*, where the Newton law is applied.
Finally, the new state is computed in the function *transition* using *Runge-Kutta 4th method*.
One of the advantage of using JAX is the *jax.jit* transform, which performs just in time compilation of a JAX python function so it can be executed efficiently. In this case it is applied to the RK4 method because it holds the constraints forced by JAX and makes the code faster.
Lastly, here it's also implemented the contact model for the dynamic of the system. In general, when an object bounces on a surface, it produces a discontinuity in the differential equation that describes its dynamic, since either the position or the velocity along some axis change.
There are two options to deal with these:

- Time stepping/Contact-implicit formulation, which solves a constrained optimization problem at every time-step and enforces no interpenetration between objects by solving for contact forces;

- Event-based/Hybrid formulation, which integrates the ODE while checking for contact events by making use of a **guard function**, for example $z \geq 0$ for the contact with the ground. In this way, when contacts happen, it executes a *jump map* that models the discontinuity and then it continues with its dynamic.

In control theory, *hybrid formulation* is easier to apply with standard algorithms such as DDP. The downside is that requires pre-specified contact mode sequences, i.e. which part of the robot is in contact at each time-step. However, with the hybrid formulation, contact forces are not explicitly computed and we can use high-accuracy integrators.
Thus, in the rope model, when a transition is performed, the new state is checked to see if some of the masses is either on the ground or it is bumping against the box where it is supposed to lay on.

3.5.2 Constraints

The file constraints.py contains two different classes for two different constraints: the sphere constraint and the box constraint, that are shown in Figure 3.5. The code is shown in the appendix section A.4.
The sphere constraint represents the 3D space where the actuated mass can actually move, i.e. where the end-effector can act. Thus, it is an inequality constraint in the position of the actuated mass.
In order to compute the needed matrices for the algorithm, the methods of this class have also to evaluate the value of the constraints for all the affected states of the trajectory. In this sense, here a scalar value will be returned and it represents the distance between the surface of the sphere and the state of the actuated mass: if it is inside the sphere is negative, otherwise positive. This follows the Augmented Lagrangian method: it will have relevance only if the mass is outside the sphere, i.e. positive, and the correspondent multiplier will be different form 0.
The sphere is constructed such that the center is in (0,0,4), position in space of the manipulated mass at the initial state, and radius of 3.
The box constraint instead is a set of inequality constraints on all the axis. In order to evaluate it, the position of each masses is checked and if it is inside the box with a certain tolerance, it returns a positive value so that it has a relevance in the Augmented Lagrangian equation. The box is implemented such that the upper face is at $z = 3.2$.
Both classes compute also the Jacobian of the correspondent constraint as required from the algorithm.

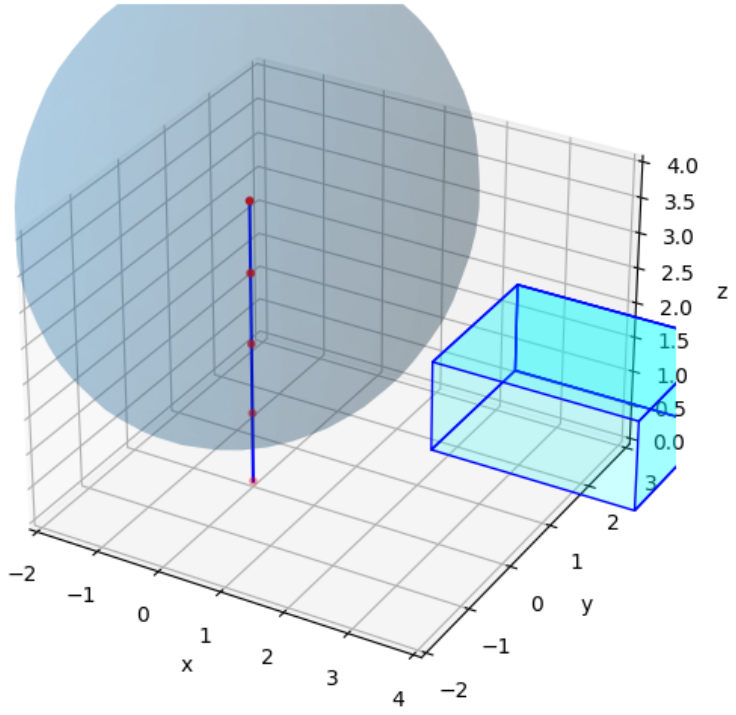

Figure 3.5: Sphere and Box constraints

Lastly, *iLQR.py* computes the *backward* and *forward* steps shown in Algorithm 4 and 6 (Appendix Section A.3), whilst *AL_iLQR.py* implements the whole algorithm as shown in Algorithm 2 (Appendix Section A.2).

Chapter 4

Analysis and Results

This section aims to assess the algorithm's performance, with a particular focus on identifying potential future scenarios and advancements. To achieve this, the performances of the solver will be analyzed by investigating the impact of various parameters. These parameters include the horizon length, the controller's sampling time, its ability to reject disturbances and the importance of initializing the penalty term μ effectively.
All the results, both tables and figures, show the outputs after 1 iteration of the solver AL-iLQR.
For a clearer visualization of the trajectories in output, the only trajectories displayed in the figures will be the ones of the manipulated mass and the last mass.

4.1 Tuning of the parameters of the cost function

In Section 3.3, the role of the cost function was discussed as well as the choice of its shape in order to achieve the proposed task. As shown in the equation 3.34, the parameters that can be chosen and tuned are the goal state x_{goal}, the running cost matrix for the state vector Q, the running cost matrix for the control input R and the final cost matrix for the final state Q_f.
For what concerns the final cost matrix Q_f, as previously explained, the relevant entries are the one relative to the position of the last three masses and they are set equally to 150 because a value of a lower magnitude isn't enough to produce a trajectory in output; in fact, with a value of 50, for example, the solver does not find a optimal or sub-optimal trajectory.
Therefore, in this section, the analysis on the other parameters will be made to better understand how the behavior and the optimal trajectory change based on the possible values.
In the following Table, the parameters of the performance shown in Figure 3.2 are

resumed, such that the analysis will be made by varying one of them each time.

Parameter	Description	Dimension	Value
x_{goal}	Goal State	30x1	4.2 along z axis
Q	Running Cost State Matrix	30x30	10^{-4}
R	Running Cost Control Matrix	3x3	5^{-3}

Table 4.1: Parameters' values of the cost function

4.1.1 The Goal State's Height

The first parameter to be analysed is the vector containing the goal state. The goal state vector is a column vector of dimension 30x1 containing the target values to be achieved by the state vector x representing the rope nodes' positions and velocities in the space at the last time-step. The first mass, the manipulated one, should get to the position (1,0,4.2) while the last one to (5,0,4.2), stating that the rest length of the springs is of length 1.
As described in Section 3.5.2, the box is positioned such that the upper face is at a height of 3.2. As it is treated like an obstacle to be avoided and it was implemented to not be penetrated, with a goal height near to the edge of the box the algorithm would have more difficulties in achieving the target, since it cannot control directly the trajectory of all the masses, but only of the manipulated one and the others through the dynamics of the system. Instead, a target slightly higher than 3.2 would allow a smoother behaviour.
In the next Table, the performances are shown in terms of cost, iterations and violation of the constraints with different values of the height of the goal; in the following Figures, the trajectory of the manipulated mass along with the final position of the rope is displayed.

Height along z	Total cost	Final Cost	n° of iLQR iterations	maximum violation of constraints
3.5	417.745	143.444	7	4.737
3.8	631.89	188.475	5	51.277
4	521.071	153.50	8	17.108
4.2	**142.837**	**80.117**	**5**	**4.23**

Table 4.2: Variation of the final total cost based on the target position

From the results above, it can be said that for values of 3.5 and 4 the performances are comparable in terms of costs and iterations, whilst the violation of the constraints is bigger for the second value.
With a value of 3.8, the performance are slightly worse for what concerns the cost of the trajectory, but the violation of the constraints is significantly higher.
As it is possible to notice from the figures below, as the target moves closer to the edge, it becomes increasingly challenging for the control input's impulse to be effective in achieving the desired task of lifting the other end of the rope to a suitable height to pass over the upper face of the box.

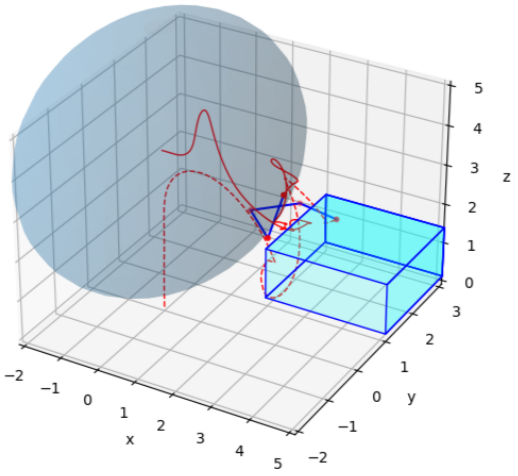

Figure 4.1: Trajectory and final position with height of 3.5

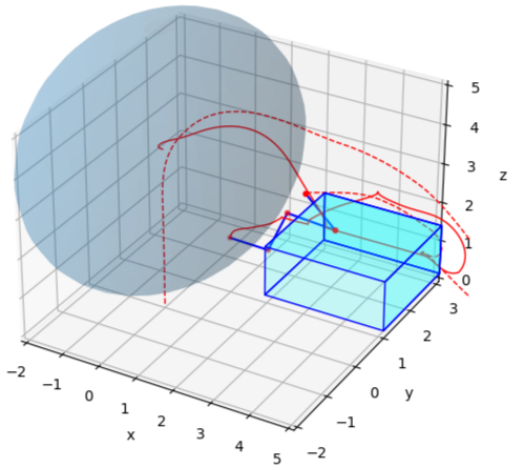

Figure 4.2: Trajectory and final position with height of 3.8

The best achieved performance is shown in Figure 4.4, even if the constraint of the radius of the sphere is slightly violated.

However, one would expect to have a similar behaviour with a height of 4; as shown in Figure 4.3, the impulse is similar to the one in Figure 4.4, but less smoother and intuitive with respect the optimal one. Looking at the maximum absolute value of the control input u, in the first case, it is around 24, whilst in the latter around 6. Due to the complex dynamics of the system and to its changing the shape as it is non-rigid, higher values of the control inputs mean greater values of external forces applied to the deformable objects, implying higher deformations and more dynamical movements of the masses, i.e. the nodes, leading to less controllable behaviours and higher total costs.

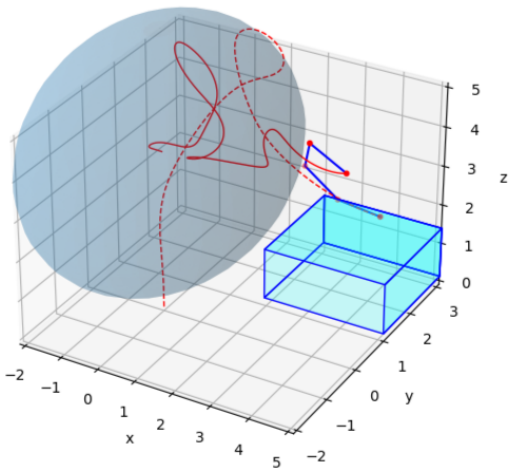

Figure 4.3: Trajectory and final position with height of 4

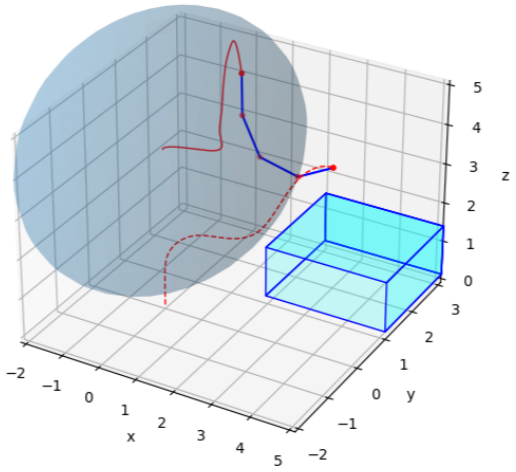

Figure 4.4: Trajectory and final position with height of 4.2

A common behaviour shared by the the first three simulation is that when the sphere constraint is violated, the solver works such that the manipulated mass is brought back inside its volume. This involves sudden changes in the direction of the control input and higher values for it, that cause deeper deformations and uncontrollable behaviour of the manipulated system along the whole trajectory, as explained earlier.
In general, a higher value with respect to the surface allows to have a smoother behaviour, thus in the following analysis the height is set to 4.2 as it gave the best performance.

4.1.2 The running cost matrices Q and R

Tuning the weight matrices Q and R, that are the *cost-to-go* or *running cost* matrices, usually involves a *trial-and-error* procedure. They are usually chosen as diagonal matrices, so that if there are n states and p control inputs, there are $n+p$ parameters to choose.
The diagonal values are chosen according to the *relative importance* of each state and control variable. As a first choice, they can be of the same order of magnitude and then they are modified to impose the desired performance. In general, if the value of the weight is high, the correspondent value of the state or control input has more relevance in the cost function so the algorithm tries to minimize it, whilst if the weight is too low, it can assume greater values.
Reminding the result achieved in Figure 3.2, that was the best in terms of trajectory and final cost, here the performance of the solver will be analysed when the values of the weights of Q and R are changed to see how it behaves accordingly. The values of Q and R for the best performance are of $Q = 1e^{-4}$ and $R = 5e^{-3}$.

Analysis on the running cost matrix Q

The first parameter to be analysed is the running cost matrix Q, while R is kept fixed to $5e^{-3}$.
The running cost Q matrix is a positive semi-definite matrix that defines the state cost, i.e. it is used to specify the relative importance of each state variable in the cost function along the time-steps from 0 to $k = N - 1$, that is along the whole trajectory except for the last state. Its relevance plays the role of trying to minimizing at every time-step the distance between the current position and the desired one.
In the following Table, the results are resumed.

Q	**Total Cost**	**Final Cost**	**n° of iLQR iterations**	**maximum violation of constraints**
0.1	11285.992	3247.118	5	-1
0.01	4617.263	48.317	8	7.551
0.001	1288.896	805.413	7	15.21
0.0001	**142.837**	**80.117**	**5**	**4.23**
0.00001	546.517	451.425	6	8.234

Table 4.3: Variation of the final cost based on the values of Q for both velocities and positions

From these results, the best outcome in terms of achieving the goal is the one with $Q = 0.01$, because the final cost is lower that the one with $Q = 0.0001$; however the violation of the constraints is higher.
The value of the total cost cannot be used as parameter to compare the performances here, because the accumulation over the time-steps is higher due to the higher weight itself. Thus, to have more information, one should look at the trajectories in output that are shown in the following Figures.

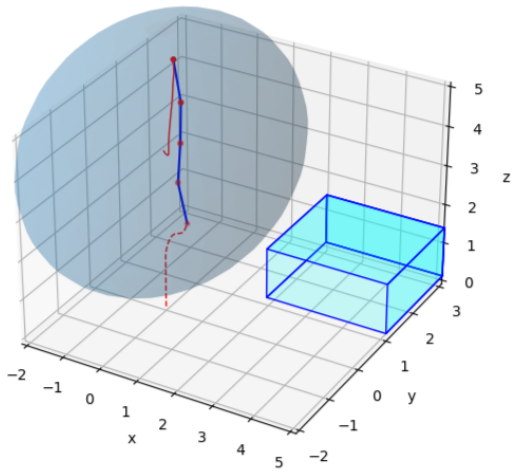

Figure 4.5: Trajectory and final position with Q = 0.1 and R = 0.005

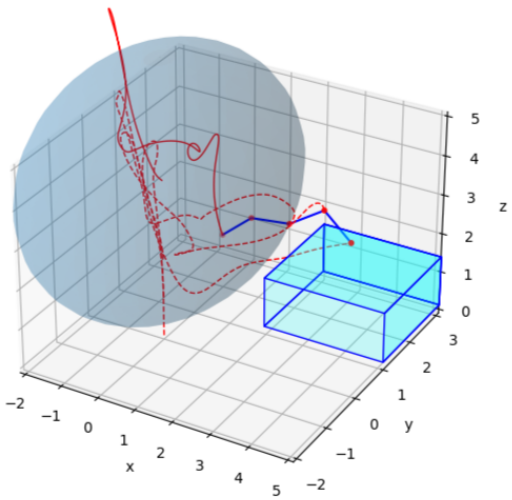

Figure 4.6: Trajectory and final position with Q = 0.01 and R = 0.005

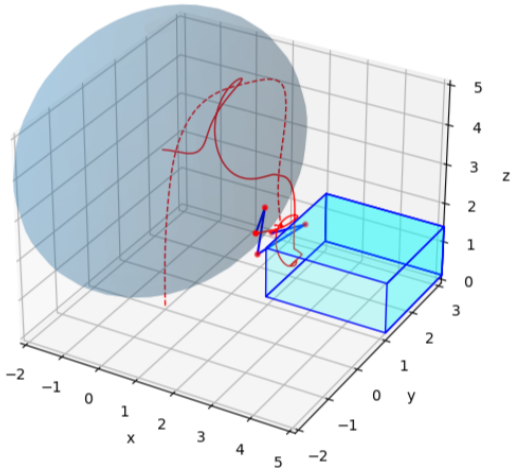

Figure 4.7: Trajectory and final position with Q = 0.001 and R = 0.005

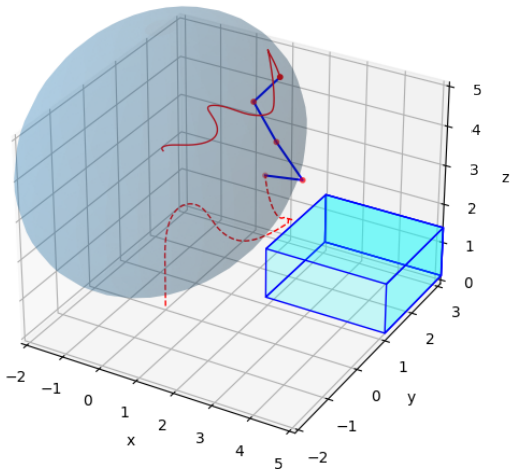

Figure 4.8: Trajectory and final position with Q = 0.00001 and R = 0.005

First, by comparing Figure 4.6 and 3.2, one can say that the trajectory in output from the latter is still the best solution in terms of overall performance, because of its smoothness and satisfaction of the constraints, as well as feasibility of the trajectory itself.
Then, it can be stated that with a high value of Q, the velocity is highly penalized, leading to smaller values than the one needed to fulfill the task and give the necessary impulse to complete the task. This can be appreciated in Figure 4.5.
Conversely, a smaller value leads to the possibility of having higher velocities and more dynamical movements, as shown in Figure 4.8.
These results take to the first of conclusion of having different weights for positions and velocities, such that one can enforce a strong fulfillment of the target through a higher value for the positions of the masses while neglecting the velocities through which the input tries to complete the task.
In this sense, for example, one could expect a different behaviour with respect the output in Figure 4.5, since the velocity is not limited as earlier.
Hence, in the following Figures the Q is set such that the velocities have a weight of $1e^{-4}$ whilst the positions have respectively {0.1,0.01,0.001,0.00001}. In this way it is possible to notice the influence of the parameter: the higher the faster it tries to achieve the goal state but now the velocity have a negligible role in the cost function, so it is not minimized to small values.
In the following Table the results are summarized.

Q	Total Cost	Final Cost	n° of iLQR iterations	maximum violation of constraints
0.1	12066.317	554.37	6	61.713
0.01	5229.63	541.328	9	20.199
0.001	502.158	293.884	7	5.359
0.0001	**142.837**	**80.117**	**5**	**4.23**
0.00001	520.071	417.326	8	9.638

Table 4.4: Variation of the final cost based on the values of Q for the position only

As expected, with respect to the results showed in Table 4.3, the final cost for the simulations with $Q = 0.1$ and $Q = 0.001$ is improved due to the fact that the velocity is not minimized anymore, whilst with $Q = 0.00001$ it remains the same, since the magnitudes of the weights of both velocity and position are comparable and the output should not change significantly.
Looking at the maximum absolute velocities, in fact:

- with $Q = 0.1$, the value goes from 3.8 to 51.2;
- with $Q = 0.001$, the value goes from 12.5 to 16.76.

To compare this with the simulation with $Q = 0.0001$, here the maximum absolute value is around 6.5.
From the results in output with $Q = 0.1$ and $Q = 0.001$, it is possible to appreciate that with a higher value of Q the solver tries to minimize in a faster way the distance from the goal, so it results in higher results and worst performances.
For $Q = 0.01$, the value of the final cost significantly increased. This could be due to the fact that in the previous configuration shown in Figure 4.6, the velocity was limited to lower values but not as in Figure 4.5. This allowed to have velocities high enough to fulfill the task. i.e. the trade-off between positions and velocities allowed to have a good result.
In this case instead, the velocity is not limited anymore, while the weight on the position is still high; hence the solver tries to achieve the target in the fastest way which leads to high velocities, high external forces and, as said earlier, less controllable shape and deformations of the deformable object.
By comparing the maximum value of the velocity in both configurations, indeed, in Figure 4.6 it is around 33, whilst in Figure 4.10 it is around 52. In the following Figures, the trajectories in output are shown.
As one could see comparing Figures 4.5 and 4.9, the first part of the trajectory is

the same in both situations, but the possibility of having a wider range of velocities, when only the position has a higher weight, takes to larger and more dynamical movements. This produces a significant violation of the constraints and a less smooth behaviour of the input, as it tries to reduce as fast as possible the distance between starting and goal positions.
By comparing the trajectories in Figures 4.6 and 4.10, one could notice how in the second configuration the velocity is not limited to a small range of values. This, however, leads to wider movements and worst behavior of the overall performance. As proof, one could try to increase by 10 the weight of the velocity to see how the solver behaves. One may expect to have a smoother behavior and better results given all the considerations said up to now.

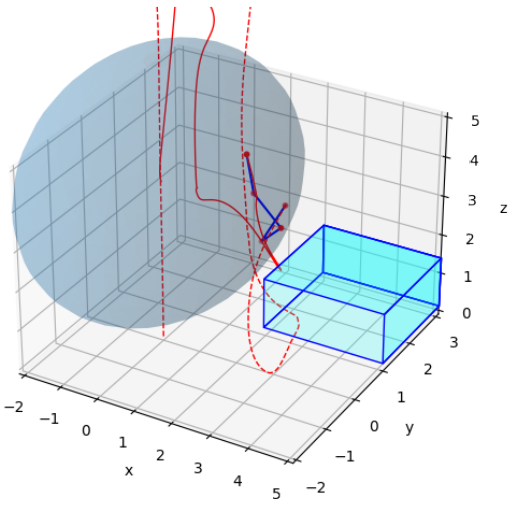

Figure 4.9: Trajectory and final position with Q = 0.1 for position only and R = 0.005

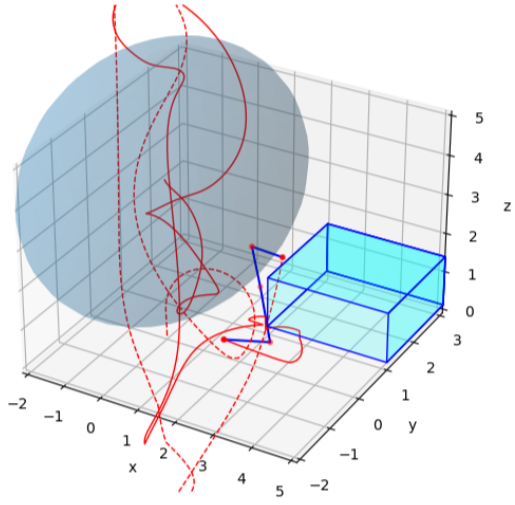

Figure 4.10: Trajectory and final position with Q = 0.01 for position only and R = 0.005

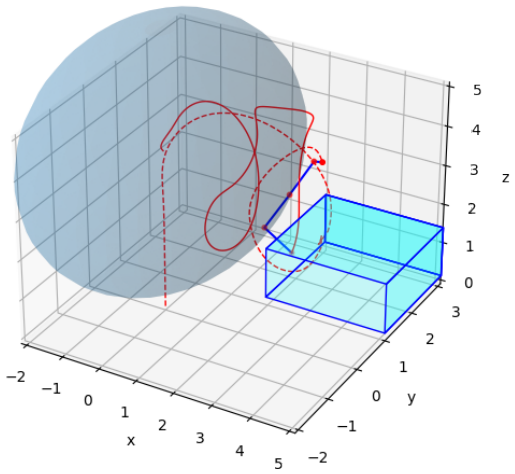

Figure 4.11: Trajectory and final position with Q = 0.001 for position only and R = 0.005

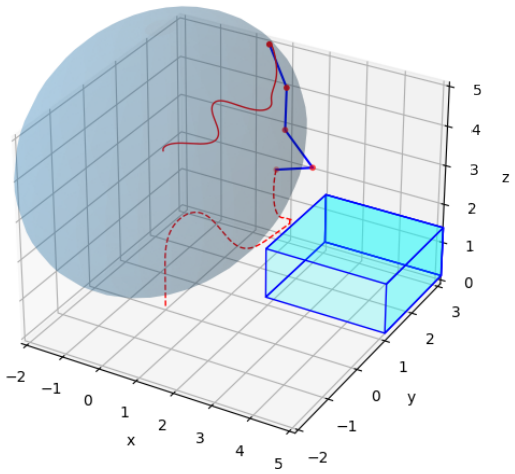

Figure 4.12: Trajectory and final position with Q = 0.00001 for position only and R = 0.005

Looking at the Table 4.4, the improvement for the value of $Q = 0.001$ is significant in terms of final cost and violation of the constraints, however the trajectory is not as smooth as with the value of $Q = 0.0001$.
Lastly, for what concerns the trajectories in Figures 4.8 and 4.12, the results are almost the same, as one may expect.

Analysis on the running cost matrix R

The running cost matrix R is a positive definite matrix that defines the control cost, also known as the control weighting matrix, used to specify the relative importance of each control input in the cost function. It determines how much weight is given to the control inputs in the control objective.
Higher values in R indicate that control inputs are more important in the control objective, while lower values imply less importance. The diagonal elements of R represent the individual weights for each control input, and the off-diagonal elements indicate the relative coupling between control inputs.
A higher weight in R makes the controller prioritize minimizing the control effort, leading to smoother and potentially less aggressive control actions. Conversely, a lower weight in R places more emphasis on controlling the states, allowing the controller to be more responsive to state deviations at the expense of potentially larger control efforts.
In this work, the maximum absolute value for the control inputs has a crucial

importance, since it is supposed to be applied by the end-effector of a robotic arm that has structural and joints limits. However, there are multiple ways of limiting the control input, either by tuning the R matrix or by using constraints on the control inputs or by using control bounds and clipping the values to the maximum and minimum when it exceeds.
As one can notice from the Table below, the control input values for the trajectory in output shown in Figure 3.2 are reasonable and low. In this section, the performance and behavior will be analyzed by adjusting the R matrix to lower or higher values compared to those used for the best outcome.
In the following Table, the results by varying the value of R while keeping the Q fixed to $1e^{-4}$ are resumed.

R	**Total Cost**	**Final Cost**	**n° of iLQR iterations**	**Min/Max control input**
0.5	6212.921	6204	1	0.001/0.001
0.05	6212.921	6204	1	0.001/0.001
0.005	**142.837**	**80.117**	**5**	**5.883/-0.982**
0.0005	1667.596	716.891	9	52.109/-19.935

Table 4.5: Variation of the performances based on the values of R

As it is possible to see from the data in the Table, with a R matrix built as an identity matrix with values on the diagonal of 0.5 and 0.05, the solver does not succeed in finding any optimal trajectory. The control input values are too low to manipulate the rope with the needed strength. One could have expected these results since with a value of 0.005 the maximum absolute value is around 6.
One could expect also the values with $R = 0.0005$, due to the fact that the weight is lower so they can assume larger values. However this lead to more dynamical movements and larger deformations, that makes the deformable object to be less controllable.
These considerations are confirmed by the trajectories in output shown in the following Figures.
As said earlier, in Figures 4.13 and 4.14, there is no trajectory in output, while in 4.16 the solver tries to fulfill the goal with more relaxed boundaries for the control input u. This results in a less controllable behavior of the non-rigid system and deeper deformations.

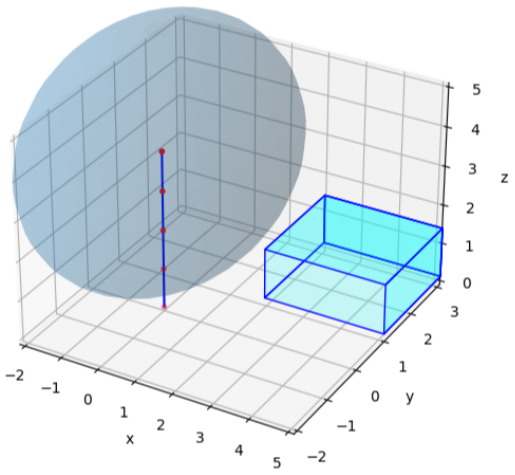

Figure 4.13: Trajectory and final position with Q = 0.0001 and R = 0.5

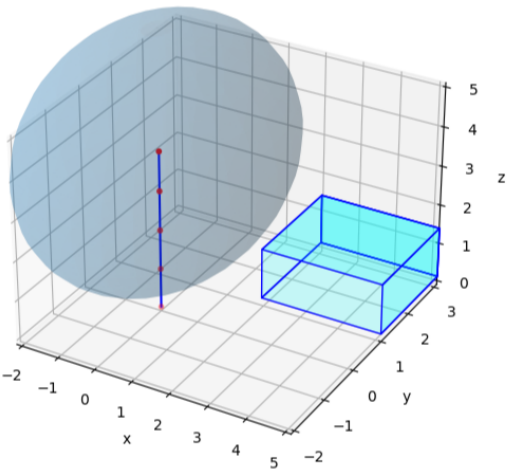

Figure 4.14: Trajectory and final position with Q = 0.0001 and R = 0.05

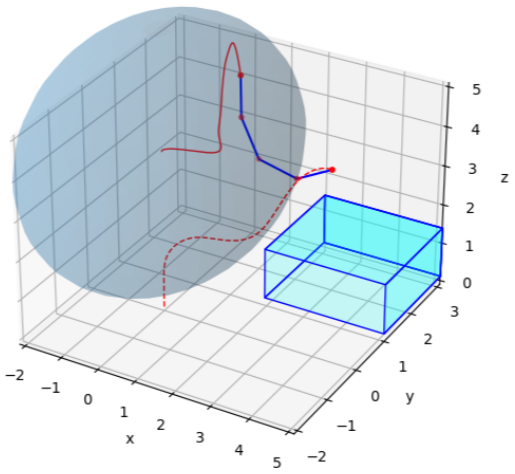

Figure 4.15: Trajectory and final position with Q = 0.0001 and R = 0.005

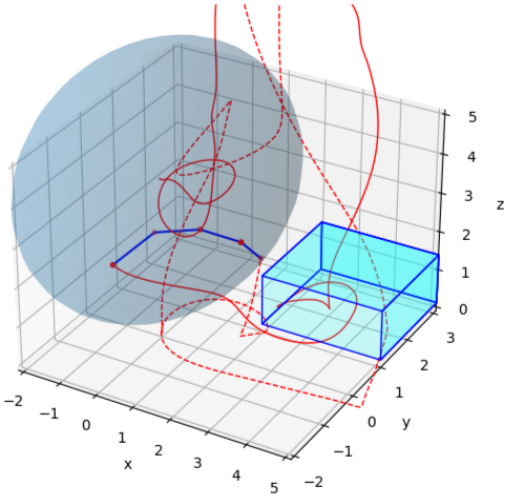

Figure 4.16: Trajectory and final position with Q = 0.0001 and R = 0.0005

The results on the analysis of the cost function lead to the performance shown in Figure 4.15 and the parameters shown in Table 4.1 at the beginning of this section.

4.2 Analysis on the horizon

In Section 3.4, the choice of the time-step for the integration time of the dynamics system of the non-rigid object was explained, along with the consequences on the performance of the algorithm. According to that, then, the forward pass of the iLQR was modified to have a lower control horizon with respect to the one needed for the simulation, in order to improve the performances with a lower computational cost required by the solver.
This section focuses on the importance of the horizon and its influence on the performance of handling a deformable object. It also discusses the implications of the horizon in terms of prediction accuracy and computational cost.

4.2.1 The role of the horizon in Optimal Control

In the field of optimal control theory, the *horizon* refers to the time duration over which the control problem is considered and it affects the outcome.
The length of the horizon can vary depending on the specific problem and application. It can be finite or infinite, and it can be discrete or continuous in time.
For trajectory optimization problems, a finite horizon is the most suitable choice because it allows to work with algorithm such DDP. Furthermore, a finite horizon reduces the computational cost by restricting the optimization problem to a finite time interval, especially for systems with complex dynamics or high-dimensional state and control spaces. Moreover, by using a finite horizon, at each optimization iteration, the trajectory can be re-optimized based on the most up-to-date information, allowing the system to respond and adapt to current environmental or task requirements.
It is also needed to keep in mind that models used for trajectory optimization are typically simplifications of the real system dynamics and, as a result, there are inherent modeling errors and uncertainties. In this sense, a finite horizon helps mitigate the impact of modeling errors by limiting the propagation of these errors over time. By re-optimizing the trajectory periodically, the system can correct for any deviations from the planned trajectory caused by modeling inaccuracies.
In addition, by optimizing the trajectory over a shorter time period, the solver can focus on achieving desired objectives within a limited duration while accounting for uncertainties and disturbances. This approach allows for more agile and responsive control, achieving optimality while ensuring robustness.
In general, a horizon that is too short may not capture the system's behavior adequately, leading to suboptimal control performance, while a horizon that is too long may introduce unnecessary computational complexity or become sensitive to modeling errors.
Summarizing these considerations, the horizon could affect the solution in terms of:

1. **Performance**: a longer horizon allows for better long-term planning and optimization, potentially leading to improved performance. By considering a larger time span, the control system can account for future changes, disturbances, and desired system behavior. On the other hand, a shorter horizon may sacrifice long-term performance in favor of more frequent updates and responsiveness to immediate changes;

2. **Computational Complexity**: longer horizons generally require more computational resources and time to solve, as they involve larger optimization problems. In practical applications, the computational complexity of solving the control problem within the given horizon may impose limitations on the real-time implementation of the control system. Therefore, there is often a trade-off between horizon length and computational feasibility;

3. **Robustness**: a longer horizon allows the control system to consider a wider range of possible future scenarios and adapt the control strategy accordingly. This increased lookahead can enhance the robustness of the control system by accounting for uncertainties in system dynamics, disturbances, or model inaccuracies. Shorter horizons, however, may limit the ability to effectively respond to uncertainties, potentially leading to sub-optimal performance in the presence of disturbances.

Hence, the choice of the horizon for the controller depends on the specific requirements of the task, the complexity of the deformable object, and the capabilities of the control system. It often requires iterative experimentation and tuning to find the optimal balance between reactivity and planning for effective manipulation of the deformable object like a rope.

4.2.2 Analysis of the outcomes based on the value of the horizon

For the reasons explained above, in this Section, the behavior and performance of the solver will be examined, focusing on the analysis of trajectories and outputs with different horizon lengths. The simulation intervals between two consecutive control updates will be maintained at 10.
In the Table below, the performances are summarized in terms of total and final cost, number of required iteration for the LQR to converge and maximum value for the violation of the constraints along the horizon.

Horizon	Total Cost	Final Cost	n° of iLQR iterations	Maximum violation of constraints
1600	818.96	718.77	5	6.13
1800	612.406	431.193	8	5.61
2000	**142.837**	**80.117**	**5**	**4.23**
2250	1225.78	717.814	7	11.689
2500	1677.602	780.93	6	19.316
2750	870.203	266.296	7	20.716
3000	1390.733	951.125	7	21.93
4000	2011.663	604.653	4	18.84

Table 4.6: Variation of the final cost based on the values of the horizon

As the total cost accumulates over the time and strictly depend on the length of the horizon, it cannot be considered as parameter to analyze the performance of the solver. Instead, the final cost and the violation of constraints are more suitable, while the number of iterations of the iLQR ranges from 4 to 8, so it's quite comparable among all of them.
One could notice that the violation of the constraints increases with the value of the horizon, which could suggest using shorter values for better performances. However, it is evident from the final cost values that an excessively short horizon is not suitable as well. Indeed, the final cost for an horizon of 1600 is comparable with the one of 4000.
In the following Figures, the trajectories in output for each of them are shown.

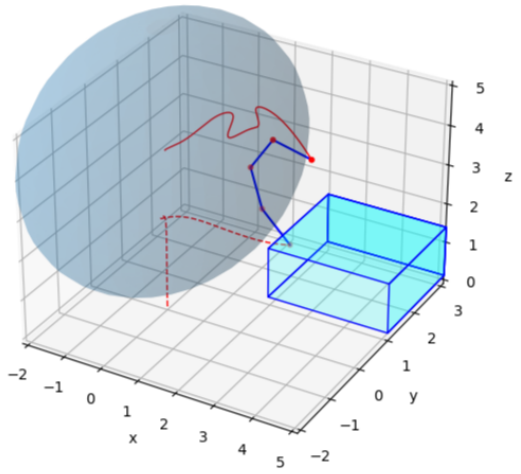

Figure 4.17: Trajectory and final position with Horizon=1600

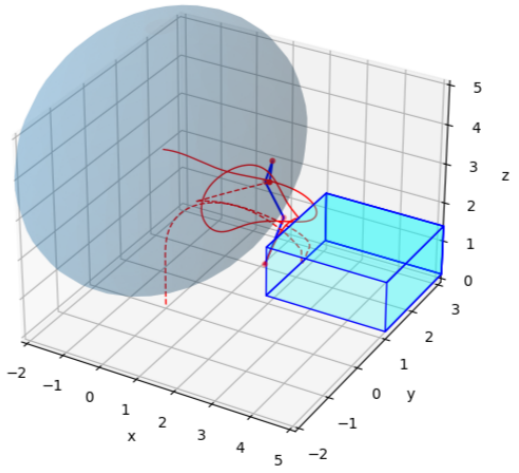

Figure 4.18: Trajectory and final position with Horizon=1800

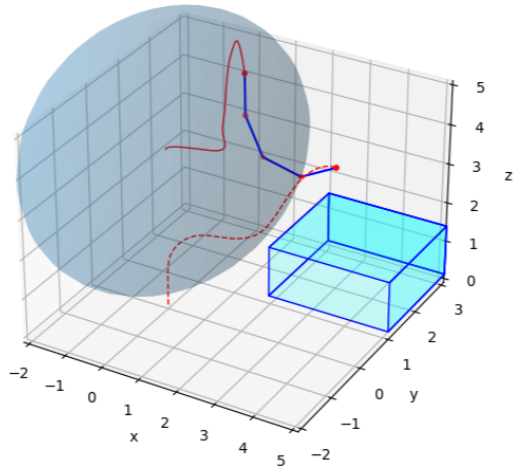

Figure 4.19: Trajectory and final position with Horizon=2000

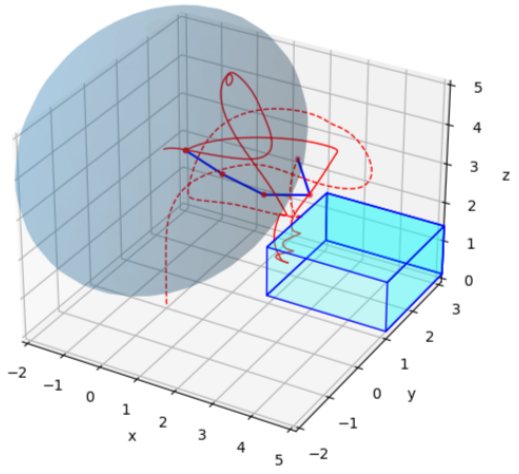

Figure 4.20: Trajectory and final position with Horizon=2250

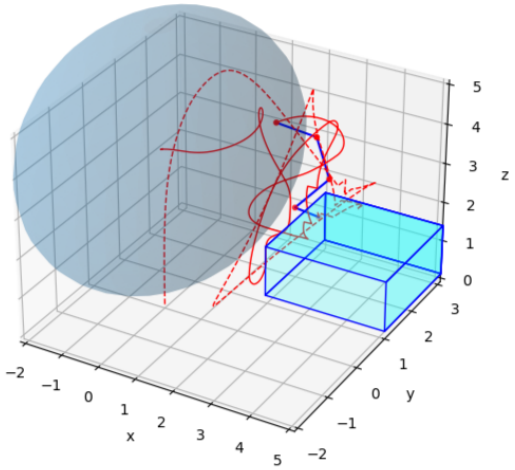

Figure 4.21: Trajectory and final position with Horizon=2500

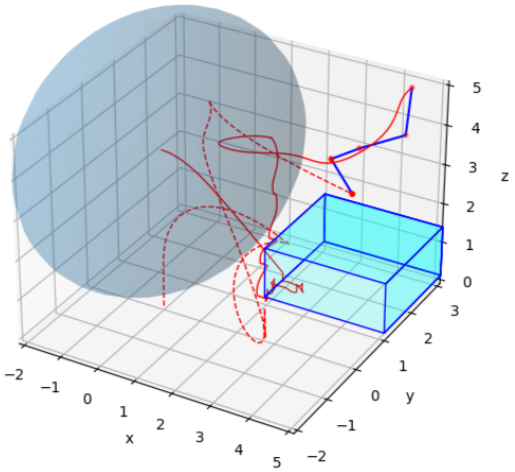

Figure 4.22: Trajectory and final position with Horizon=2750

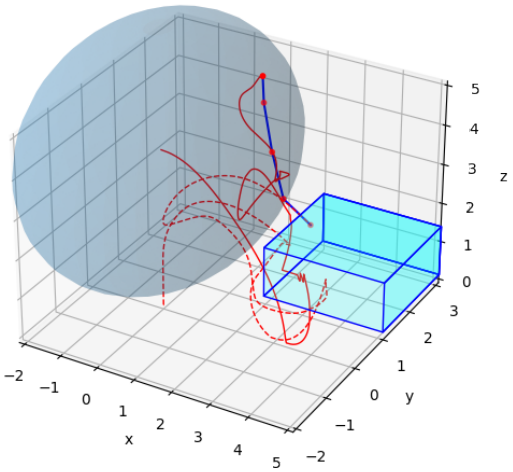

Figure 4.23: Trajectory and final position with Horizon=3000

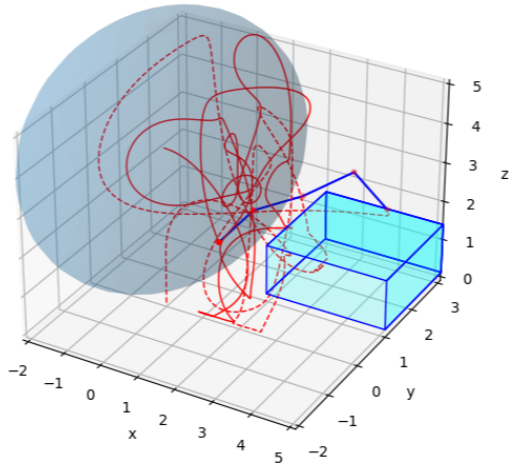

Figure 4.24: Trajectory and final position with Horizon=4000

As it is possible to appreciate from the Figures above, with horizons of 1600 and 1800 in Figures 4.17 and 4.18, the trajectory is quite smooth and they almost achieved the target, however they are sub-optimal with respect to the trajectory shown in Figure 4.19, that is the most successful one among all of them in terms

of final cost and violation of the constraints. The observation that can be made is that this behaviour of the trajectory, that gets worse as long as the length of the horizon increases, can be related to task itself. In fact, since the time-step for the system simulation is of 0.001, having a length of the horizon of 2000 means having a simulation time for the task of 2 seconds.
The results shown above suggest that the length of the horizon could be dependent on the task itself, which means that for this specific task a too short time (less than 2 seconds) does not allow to achieve the goal, whilst a longer time interval (more than 2 seconds) is not suitable for the task itself because the solver is trying to develop the movements along a time interval that is too long for achieving the needed dynamical behavior to reach the proposed purpose.
In this sense, the results suggest that there is an **optimal simulation time** for this specific task, i.e. the dynamical manipulation that the solver is trying to achieve should be fast enough: the trajectory shown in Figure 4.19 is the dynamical movement that this work is trying to pursue and doing it in 2 seconds allows to have the suitable control inputs and velocities to realize it. Instead, when attempting to accomplish the task within a longer timeframe, such as 3 or 4 seconds, that trajectory undergoes a deceleration. Consequently, this limitation prevents the attainment of the appropriate impulse and velocities, resulting in an output that falls short of achieving the intended purpose.
The relation between the final cost and the horizon length is summarized in Figure 4.25.

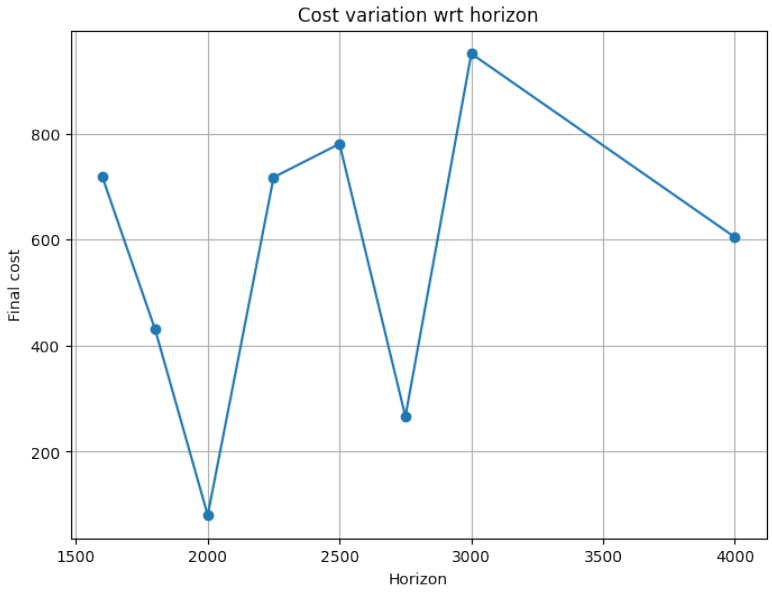

Figure 4.25: Variation of the final cost vs the horizon

Based on the numbers, with the horizon of 2000, the solver shows also a faster convergence with 5 required iterations and a better satisfaction of the constraints with a value of 4.3.

4.2.3 Analysis on the sampling time of the controller

The time-step of a controller is another important parameter to consider, and it can impact the performance and stability of the control system. The time-step of a controller refers to the interval at which the controller computes its output based on the system input and feedback.
Here it is possible to analyse this parameter because it is a simulation, otherwise the parameter is fixed based on the frequency of the controller. Indeed, in a digital control system, the time-step is determined by the sampling rate of the analog-to-digital converter (ADC) that measures the system input and feedback, and the processing time of the micro-controller or digital signal processor (DSP) that runs the control algorithm.
If the time-step of the controller is too large, the controller may not be able to respond quickly enough to changes in the system input or disturbances, leading to unstable behavior. On the other hand, if the time-step is too small, the controller may consume too much processing power or memory, leading to computational delays or overshoot. The appropriate time-step of a controller depends on several factors, such as the dynamics of the controlled system and the computational resources available. In general, a smaller time-step can provide better control performance by allowing the controller to respond more quickly to changes in the system, but it may also increase the computational demands of the controller.
Therefore, a balance between performance and computational efficiency needs to be struck when choosing it.
In practice, the optimal time step is often determined by trial and error, starting with a relatively large time step and gradually decreasing it until the desired level of accuracy is achieved, or until the computational cost becomes prohibitive. The choice of the time step may also depend on the specific requirements of the application, such as real-time constraints or safety considerations.

4.2.4 Analysis of the outcomes based on the value of the frequency of the controller

In the previous sections, the best result was obtained with a controller's sampling interval of 10 over a simulation horizon of 2000, which leads to a simulation time of 2s. In this section instead, the behavior and performance of the solver based on the sampling interval of the controller is analyzed.
As said earlier, the length of the simulation horizon is important in terms of

dynamics of the task and the trajectory in output; therefore the length of the horizon will be fixed to 2000, while the analysis will be made by changing the sampling interval of the controller.
In the Table below, the performances are summarized in terms of total and final cost, number of required iteration for the LQR to converge and maximum value for the violation of the constraints along the horizon.

Frequency	**Total Cost**	**Final Cost**	**n° of iLQR iterations**	**Maximum violation of constraints**
0.002	3238.56	542.53	6	13.86
0.004	1674.39	771.68	5	16.52
0.005	4719.69	3063.79	3	34.32
0.008	614.32	382.27	11	6.02
0.01	**142.837**	**80.177**	**5**	**4.23**
0.012	333.91	160.04	6	-0.39
0.014	3674.43	3475.57	3	15.21
0.016	3930.54	3591.64	3	21.11

Table 4.7: Variation of the final cost based on the values of the time-step

From the values in the Table above, the best results in terms of number of violation of the constraints and final cost of the trajectory are obtained with simulation intervals of 8-10-12.
With a sampling time of 0.002 and 0.004, the final cost and violation of constraints are comparable whilst the total cost of the first is significantly higher even though the simulation horizon is the same; this means that the output may assume a worst behaviour.
Instead, if the time-step interval is too long as in the case of 14 and 16, the solver produces an output where the total and final cost are comparable. This may suggest that the solver did not produce any useful trajectory, due to the fact that it failed to capture the dynamics of the deformable object, due to the long interval between two samples.
In the Figures below there are shown the correspondent trajectories in output.

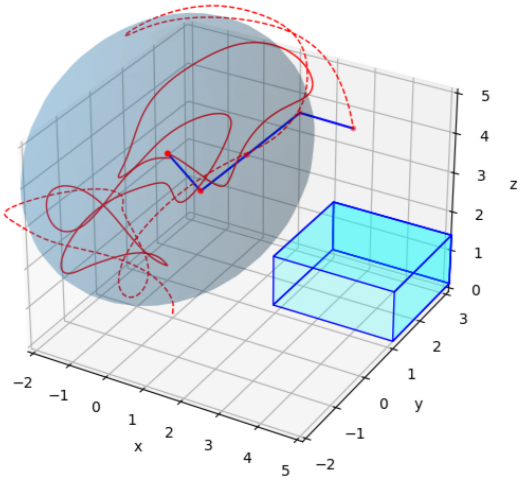

Figure 4.26: Trajectory and final position with control input update every 2 simulation steps

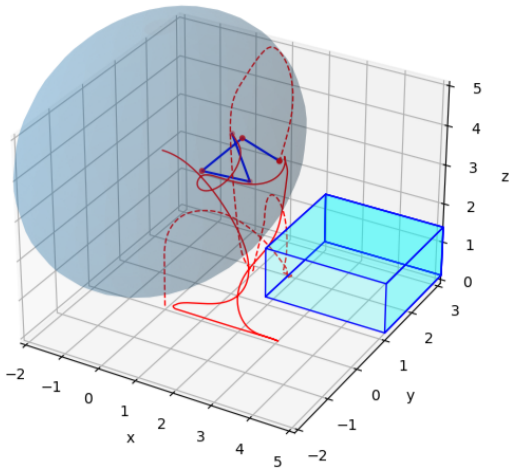

Figure 4.27: Trajectory and final position with control input update every 4 simulation steps

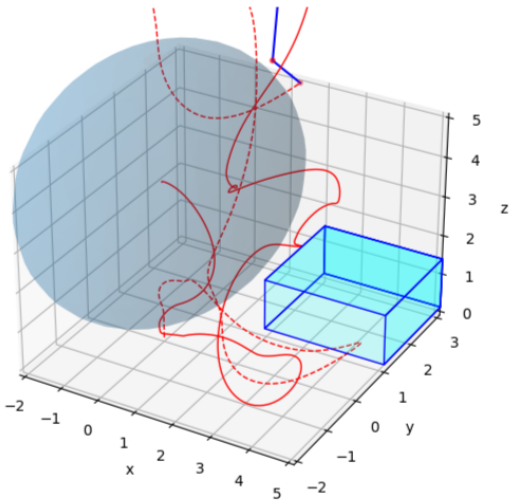

Figure 4.28: Trajectory and final position with control input update every 5 simulation steps

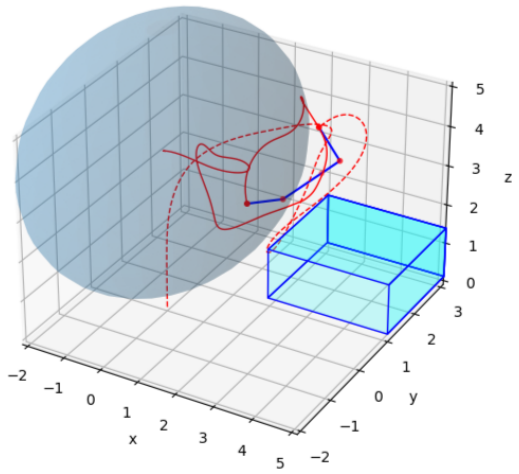

Figure 4.29: Trajectory and final position with control input update every 8 simulation steps

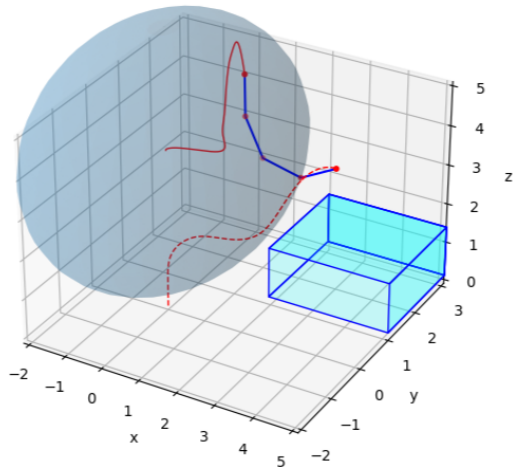

Figure 4.30: Trajectory and final position with control input update every 10 simulation steps

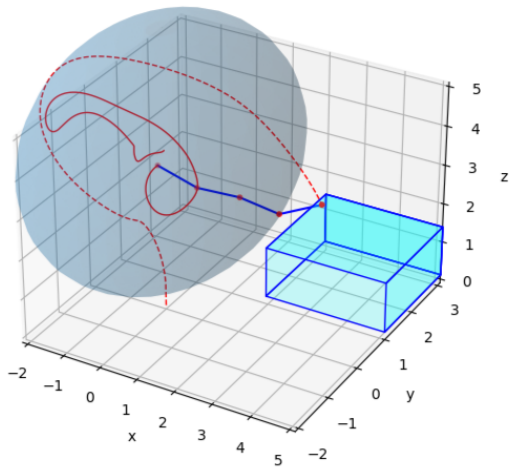

Figure 4.31: Trajectory and final position with control input update every 12 simulation steps

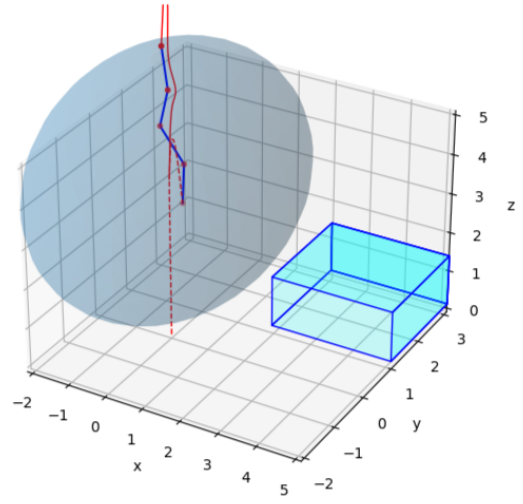

Figure 4.32: Trajectory and final position with control input update every 14 simulation steps

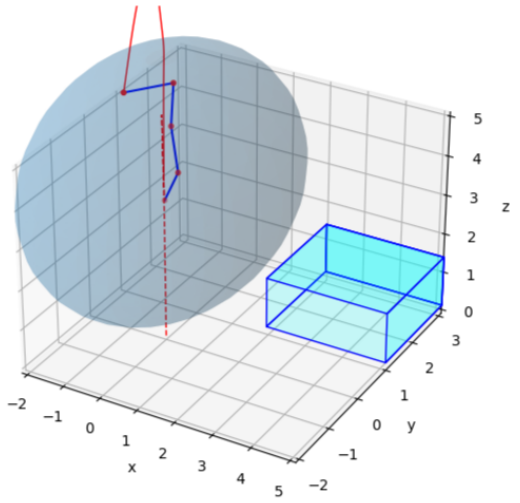

Figure 4.33: Trajectory and final position with control input update every 16 simulation steps

As shown from the trajectories in output, the best rate for the sampling is the one shown in Figure 4.30, which represents a good trade-off between length of

the simulation steps and responsiveness to the dynamical behavior of the rope in changing its shape and position along the trajectory.
Also, the trajectories in output with a time-step interval of 0.008 and 0.012 in Figures 4.29 and 4.31 achieve the target but the trajectories are sub-optimal with respect to the one in figure 4.30.
From the first three Figures it can be seen instead that if the time-step of the controller is too low, it leads to unstable and uncontrollable behavior, whereas with a time-step that is too high as shown in Figures 4.32 and 4.33, the controller is not able to capture the dynamics of the deformable objects not producing a suitable trajectory in output.
In Figure 4.34, it is shown the trends of the results in terms of final cost.

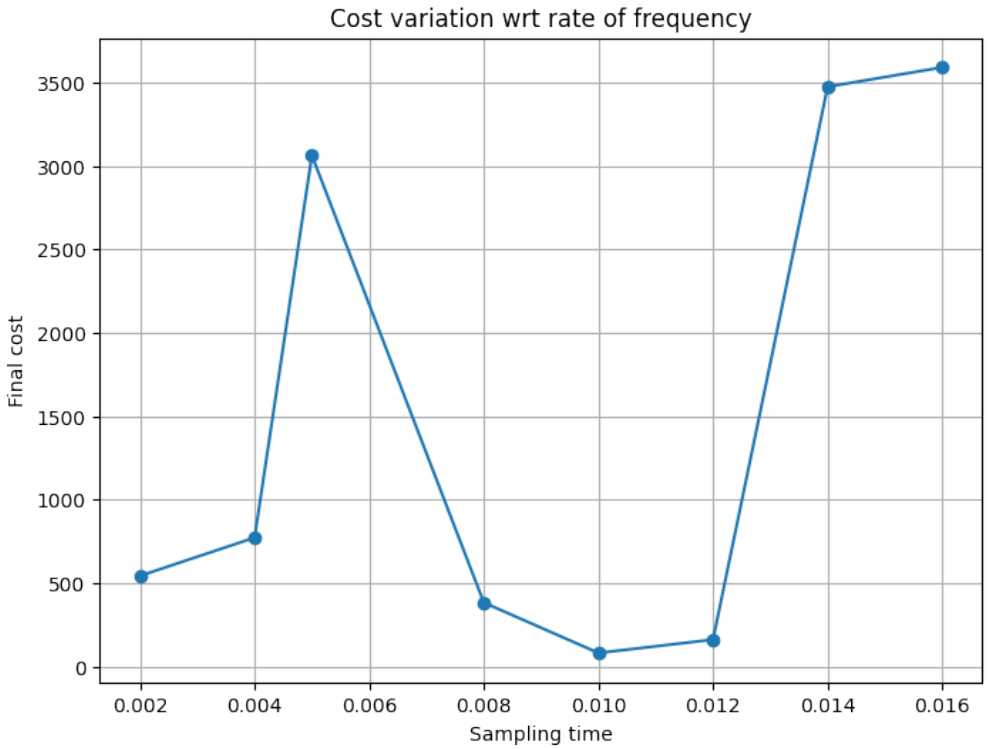

Figure 4.34: Variation of the final cost vs the time-step of the controller

Here again, it is possible to analyze the behavior of the final cost term with respect to the values of the sampling time. As shown, the lower value is achieved with the sampling interval of 10.

4.3 Disturbance rejection

The robustness of a trajectory optimization solver refers to its ability to consistently and reliably produce valid solutions across different scenarios, even in the presence of uncertainties, variations, or challenging conditions. A robust solver should be able to handle various sources of disturbances or perturbations without compromising the quality or feasibility of the optimized trajectories.
A robust solver should be capable of accounting for uncertainties in the system dynamics, environmental conditions, or input parameters. It should be able to incorporate probabilistic or stochastic models and adapt its optimization process to ensure that the resulting trajectories are feasible and effective under uncertain conditions.
Moreover, trajectory optimization typically involves a set of constraints, such as physical limitations, safety requirements, or operational constraints. A robust solver should ensure that these constraints are consistently satisfied across different scenarios, even in the presence of disturbances or variations. It should be able to handle both hard constraints (strict limitations) and soft constraints (preferences or objectives).
To test the ability of the solver to reject the disturbances, the following line of code is added to the *forward pass* shown in Algorithm 6

```
1    x=x+np.random.uniform(low=-5e-3,high=5e-3,size=(30,1))
```

This noise vector acts on the state vector x at each simulation step before the optimal control input u is updated as:

$$u = u^* + K^*(x - x^*)$$

where * indicates the optimal values of control input and state vector in output where the simulation is run without noise.

It basically adds a uniformly distributed vector of the size of the state vector x to the state itself. The distribution spans from $-5e^{-3}$ to $5e^{-3}$ or from $-1e^{-3}$ to $-1e^{-3}$.
Due to the randomness of the noise vector, the controller has been run for 100 simulation for both cases and the trajectories in output are shown below, where the blue ones are the one affected by the noise, whilst the red one is the optimal one and the one used as reference.

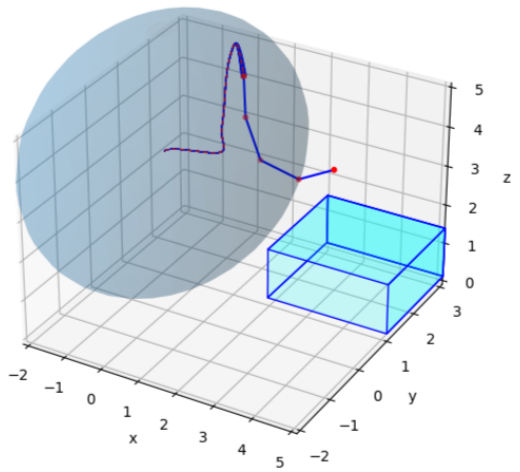

Figure 4.35: Trajectory and final position with noise normally distributed among $-1e^{-3}$ and $1e^{-3}$

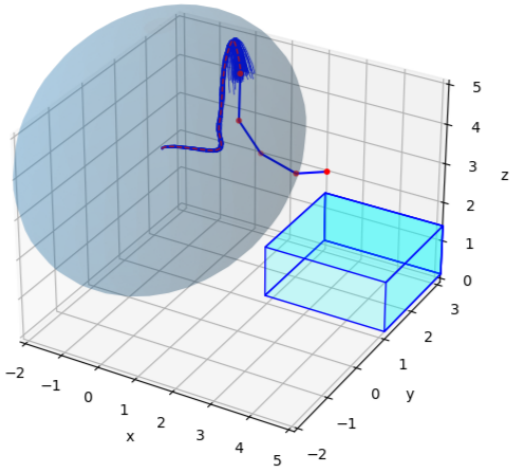

Figure 4.36: Trajectory and final position with noise normally distributed among $-5e^{-3}$ and $5e^{-3}$

As it is possible to see from the Figures above, the controller is able to handle properly the noise. The behavior at the last time steps of the horizon is due to the fact that for the last 10 instants of the simulation horizon the controller is not updated anymore while the noise is still present.

4.4 Analysis of the performance based on the initial value of the penalty term μ

When optimizing a trajectory using the Augmented Lagrangian and method of multipliers, the penalty term μ plays a crucial role in balancing the trade-off between the original objective function and the constraint violation.
Indeed, the penalty term μ affects the convergence behavior of the optimization algorithm because a small value may lead to faster convergence but can result in sub-optimal solutions with large constraint violations.
On the other hand, a large value increases the penalty for constraint violations, which may slow down the convergence but can yield solutions with better constraint satisfaction.
The choice of μ should be a trade-off between convergence speed and solution quality.
Furthermore, it helps enforce feasibility by penalizing constraint violations. As μ increases indeed, the optimization algorithm becomes more sensitive to constraint

violations, encouraging the trajectory to satisfy the constraints more strictly. However, if it is set too high, even small constraint violations may lead to excessively large penalties, causing convergence difficulties and instability.
Moreover, it determines the weight assigned to constraint violations relative to the original objective function. If it is too small, the optimizer may prioritize the original objective, neglecting the constraints. Conversely, if too large, the optimizer may excessively penalize constraint violations, leading to a significant deviation from the original objective. The value of μ should be chosen such that both the objective function and constraints are adequately balanced.
In addition, the penalty term can influence the sensitivity of the optimization algorithm to the initial conditions. Higher values can make the optimization more sensitive to the initial guess or starting point, potentially leading to different solutions for different initial conditions. Lower values instead may provide more robustness to the initial conditions but can result in convergence to sub-optimal solutions.
Finally, the value should be chosen with consideration for the scale of the problem. If the problem involves large constraints or a wide range of magnitudes in the objective function, it may need to be adjusted accordingly. Scaling the penalty term appropriately can ensure that the optimization algorithm is effective in handling both small and large-scale problems.
In summary, selecting an appropriate value for the penalty term μ is a crucial task in trajectory optimization using the Augmented Lagrangian and method of multipliers. It requires considering factors such as convergence, feasibility, constraint handling, sensitivity to initial conditions, and problem scalability. Careful tuning of μ can lead to improved convergence speed, better constraint satisfaction, and high-quality solutions.
In the previous simulations, the value for the penalty term was set to 0.1 and, as a result, the best trajectory was the one in the following Figure.

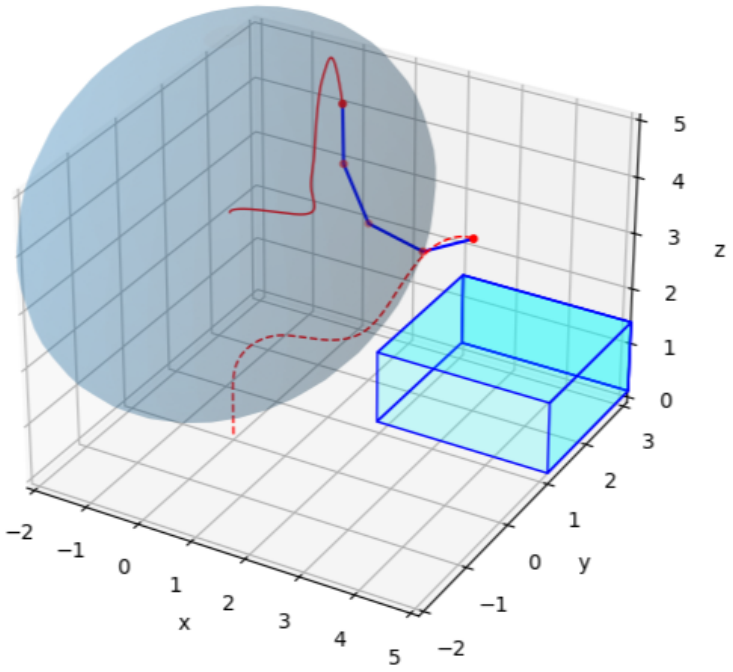

Figure 4.37: Trajectory and final position without $\mu = 0.1$

In this section, the role of the penalty term μ will be analysed. In this first part the trajectories in Figures from 4.38 to 4.43 and the results shown in Table 4.8 refers to the performance of the solver after 1 iteration of the AL-iLQR algorithm. As shown in Algorithm 2, after 1 iteration, if the violation of the constraints is not satisfied, the penalty term μ is updated by multiplying it by a constant factor that was set to 10. By increasing it, one should expect a better satisfaction of the constraints as the number of iteration grows.
Thus, in the second part it will be analysed how the performance changes along different iterations of the algorithm.
In the following Table, there are summarized the results in terms of total and final cost, number of iterations of iLQR and violation of the constraints.

μ	**Total Cost**	**Final Cost**	**n° of iLQR iterations**	**Maximum violation of constraints**
5	4024.902	3970.525	2	0.118
1	227.428	144.1705	4	2.307
0.1	**142.837**	**80.117**	**5**	**4.23**
0.01	220.24	116.01	5	6.77
0.001	727.17	640.19	5	17.36
0.0001	733.55	646.41	6	17.42
0.00001	734.35	647.198	7	17.44

Table 4.8: Variation of the final cost based on the values of the penalty term μ

As said earlier, with a large value of μ as initialization, the non-violation of the constraints is preferred to the achievement of the goal: in fact, with respect to a μ equal to 0.1, the maximum violation of the constraints is lower in the case of 1, 2.307, and even lower with an initialization of 5, 0.118. Whereas, the final cost increases to 144.17 and 3970.525 respectively.
When the value is too low instead, as it resulted with a value lower than 0.001, the relaxation of the constraint is higher, the value of the violation is around 17, but at the same time the trajectory seems to be sub-optimal.
In the following Figures, there are shown the trajectories in output.
In Figure 4.38, it is possible to appreciate how a large value can affect the convergence of the solver: the strict requirement on the constraints does not lead to any trajectories that fulfill the task.
In Figure 4.39, instead, the trajectory starts to look similar to the best one, but it is a sub-optimal solution to preserve the satisfaction of the constraints.

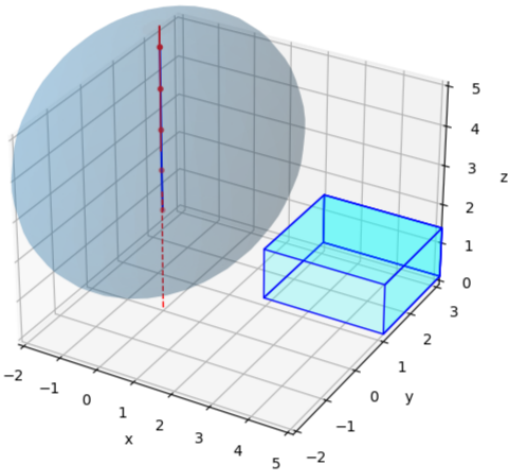

Figure 4.38: Trajectory and final position without $\mu = 5$

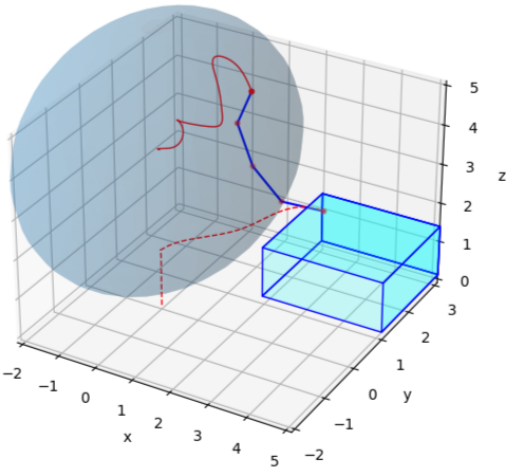

Figure 4.39: Trajectory and final position without $\mu = 1$

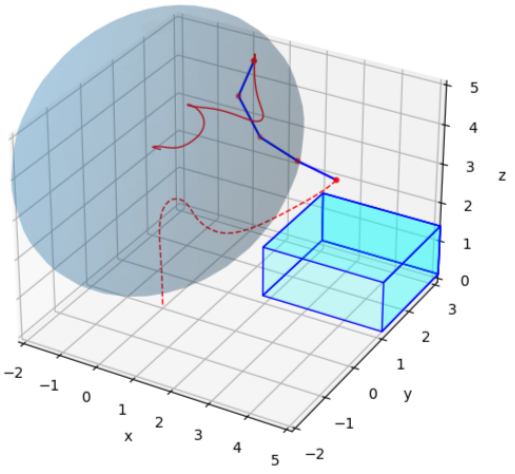

Figure 4.40: Trajectory and final position without $\mu = 0.01$

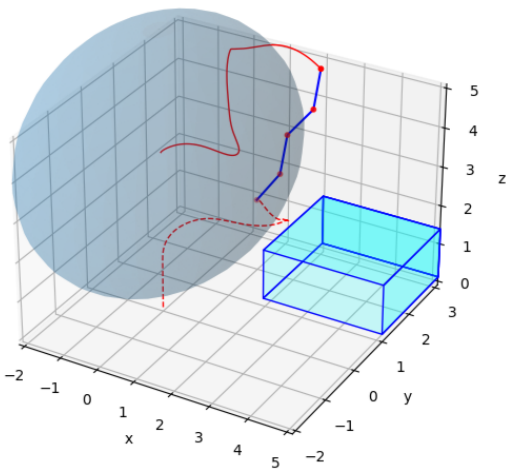

Figure 4.41: Trajectory and final position without $\mu = 0.001$

In Figure 4.40 it is possible to notice that the satisfaction of the goal is more important than the requirements of the constraints; indeed the movements are wider and the final position is getting closer to the goal one.

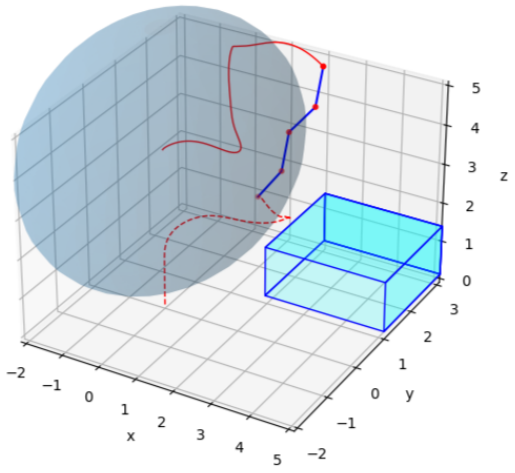

Figure 4.42: Trajectory and final position without $\mu = 0.0001$

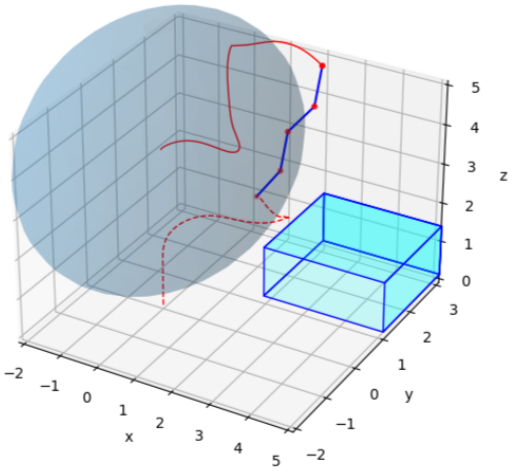

Figure 4.43: Trajectory and final position without $\mu = 0.00001$

Lastly, with a penalty term μ lower than 0.001, the trajectories in output are almost the same as it is possible to notice from both Table 4.8 and the Figures 4.41, 4.42 and 4.43. This leads to the maximum relaxation about the violation of the constraints, but at the same time the goal is not achieved as in Figure 4.37.
One might expect that with a penalty term lower than 0.01, the solver could produce better results in the next iterations of the algorithm since the requirements on the constraints become more strictly, thus the trajectory in output should get closer to the optimal one.
For this purpose, the solver was simulated again with a maximum number of iteration for the AL-iLQR of 15. The results are shown below.
With $\mu = 0.1$, after the first iteration, the solver is not able to find any better solution, even if the penalty term is increased at every iteration.
With $\mu = 0.01$, there is a slightly improvement after the first iteration as shown in the Table and Figures below.

Iteration	Total Cost	Maximum violation of constraints
1	220.2438	6.7796
2	204.2344	5.7389
3	204.2338	5.7388

Table 4.9: Improvements of the performance after the first iteration

After the first iteration the cost is again minimized, however in the next ones it remains almost the same until it does not find any better trajectory. In the Figures below the difference can be slightly perceived since the movements in the second Figure are a little bit less wide that the first one.

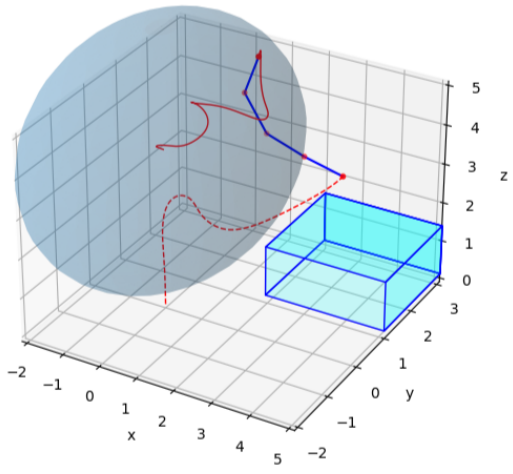

Figure 4.44: Trajectory and final position - first iteration

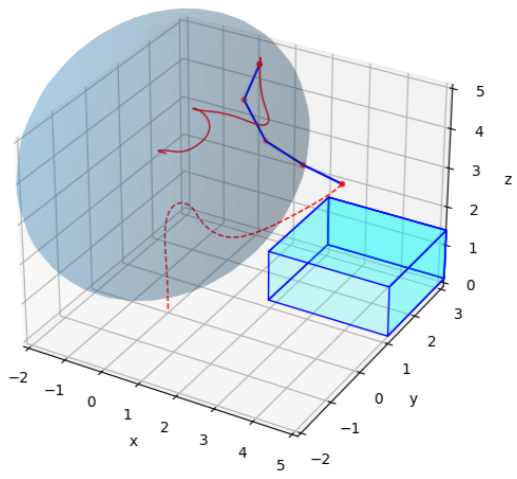

Figure 4.45: Trajectory and final position - second iteration

With $\mu = 0.001$, there is a greater improvement with respect to the previous case after the first iteration as shown in the Table and Figures below.

Iteration	Total Cost	Maximum violation of constraints
1	727.1759	17.3567
2	536.9074	5.6811
3	536.8984	5.6721
4	536.8984	5.6721
5	536.8984	5.6721
6	536.8979	5.6719

Table 4.10: Improvements of the performance after the first iteration

As it is possible to appreciate, the cost significantly decreases after the first iteration as the penalty term is update and consequently the violation of the constraints decreases as well. However, after the second iteration the improvements is almost imperceptible until it does not find any better trajectory.
Following this, there are two of the trajectories in output from the different iterations where it is possible to notice the improvement that in this case is remarkable.

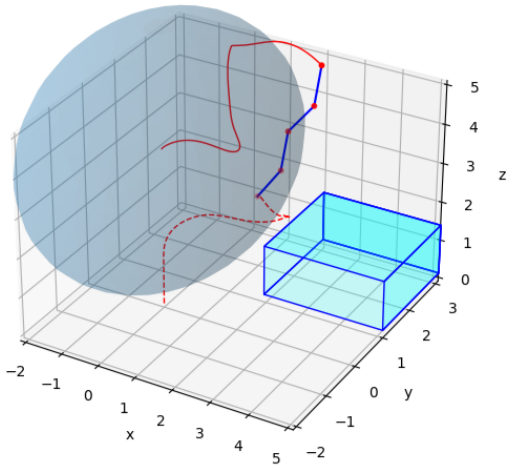

Figure 4.46: Trajectory and final position - first iteration

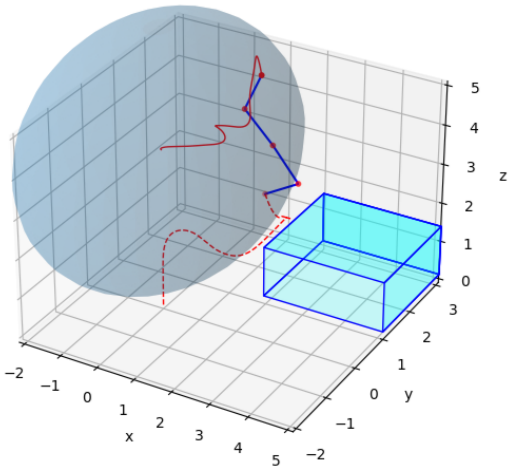

Figure 4.47: Trajectory and final position - forth iteration

In this case, the manipulated mass shows significant improvement in staying within the sphere constraint. As a result, the goal is achieved more successfully: indeed, a similarity can be observed in the trajectory in Figure 4.45 and Figure

4.47.

4.5 Discussion and limitations

This section will provide a comprehensive discussion of the overall results obtained from the algorithm and the rope model. It will analyse the advantages and disadvantages of the three main factors: the *method of multipliers*, the *iterative Linear Quadratic Regulator* and the *mass-spring-damped model* for the rope.

4.5.1 Limitation and advantages of the spring-damped-mass model for the rope

The chosen model for the deformable object, in this case, a piece of rope, is one of the primary factors influencing the overall performance. As discussed in Section 3.2, the Mass-Spring-System is a straightforward and intuitive approach for modeling non-rigid objects and deformations. The implementation utilized in this study benefits from a differentiable model, making computations easier by leveraging the built-in gradient function provided by the JAX module for NumPy and Python.
However, the simplicity of the model's implementation comes at the cost of reduced accuracy compared to other potential implementations. This is due to inherent modeling errors resulting from simplifications and approximations. A key challenge lies in parameter tuning, which in this case involved a trial-and-error procedure to set the parameters for a rope model with five nodes and a rest length of 1.
It should be noted that altering the number of nodes can introduce instability or unrealistic behavior in the simulation. Liu et al. [22] have addressed this issue by employing learning algorithms for parameter tuning.
As highlighted in Section 3.4, the discrete nature of the model introduced challenges in determining the appropriate time-step for the simulation. Time-steps higher than $1e^{-3}$ resulted in unrealistic behavior, while smaller time-steps would have incurred significant computational costs for the control algorithm and optimization solver. Consequently, modifications were made to the forward pass of the iLQR algorithm, and further analysis was conducted on the horizon length (Section 4.2.1) and the interval between two control updates (Section 4.2.3).
Overall, the model demonstrates satisfactory performance for the intended purpose. Nevertheless, it is worth noting that high control input values can lead to unstable and non-smooth trajectories.

4.5.2 Limitations of iterative Linear Quadratic Regulator

The primary issues related to iLQR examined in this work are the shape of the cost function and parameter tuning, as discussed in Section 4.1. Additionally, the

robustness of the algorithm is explored in Section 4.3. While iLQR aims to optimize the control policy, it does not explicitly address stability or robustness concerns. Manipulating deformable objects involves addressing uncertainties, model errors, and disturbances that can impact stability.
Regarding the cost function, its shape heavily depends on the specific task at hand, and the tuning of its parameters may vary accordingly. Proper parameter tuning has been shown to enable smooth achievement of the proposed target, even though it may result in sub-optimal trajectories. However, it is important to note that the algorithm is task-oriented, meaning that most parameters were tuned specifically for the goal in this particular configuration. Changing factors such as the starting position or the goal position may not yield the same level of smooth convergence, as the optimal solution found for the current setup may not be applicable.
In summary, this work emphasizes the significance of parameter tuning for achieving the desired goal smoothly. Nevertheless, it highlights the task-specific nature of the algorithm, as different configurations or objectives may require adjustments to the parameters for optimal performance.

4.5.3 Limitations and advantage of method of multipliers

To address the nonlinear constraints, the implementation utilized an augmented Lagrangian approach with a penalty term using the method of multipliers. However, a major concern associated with this approach is the penalty term, as discussed in Section 4.4, and the slow convergence of the algorithm towards satisfying the constraints. If an optimal or sub-optimal trajectory is not found in the initial iteration, the solver may struggle to converge properly. Nevertheless, the combined use of Lagrange multipliers and penalty methods offers faster convergence compared to traditional penalty methods for constrained optimization problems. By incorporating the constraints into the objective function, it transforms the problem into an unconstrained one while ensuring constraint satisfaction.
The literature demonstrates that this approach can handle various types of constraints, including equality constraints, inequality constraints, and bound constraints, providing flexibility in handling a wide range of constraints within the optimization problem.
In Section 4.4, it was revealed that the penalty parameter governs the balance between the objective function and constraint satisfaction. It enables adaptive adjustment during iterations, enhancing the convergence behavior. However, careful tuning of the penalty parameter is required to achieve desirable convergence behavior. Improper selection of the penalty parameter can lead to slow convergence or difficulties in attaining the desired solution.

Chapter 5

Conclusions

In conclusion, this thesis has examined the application of a model-based approach to address a constrained trajectory optimization problem in non-rigid object manipulation. The iterative Linear Quadratic Regulator (iLQR) method was chosen for its efficiency in solving the problem while offering a simpler implementation compared to the standard Differential Dynamical Programming (DDP) algorithm. To handle the constraints, the Augmented Lagrangian and Method of Multipliers technique was implemented. A Mass-Spring-System (MSS) model was selected to represent the non-rigid object, specifically a rope, due to its simplicity and intuitive nature. The differentiability of the model has been enabled by the JAX environment, that made possible to use built-in gradients functions.
The rope model was carefully tuned to closely replicate real-world behavior, aiming to enhance the accuracy of the final results. The entire AL-iLQR algorithm was tested and fine-tuned to achieve optimal results within the specified conditions. The solver's behavior was analyzed by varying the parameters of the cost function, horizon length, and frequency rate. Additionally, the controller was tested against noise, and an in-depth analysis was conducted on the penalty parameter μ, which strongly influenced the outcome.
The analysis demonstrated the validity of a model-based approach when a defined and differentiable model is available. However, it should be noted that the Mass-Spring-System (MSS) model, while convenient, may not provide the best accuracy. Indeed, using a Finite Element Method (FEM) solutions would be more suitable for simulating the deformable object. This could be a further improvement of this result, even though it would cause a higher computational cost. Moreover, the importance of a suitable cost function was highlighted as it significantly improved the overall performance.
The limitations of the Augmented Lagrangian and Method of Multipliers (ALMM) algorithm were discussed, particularly in relation to the influence of the penalty parameter μ. The thesis also emphasized the trade-off between simulation and

control horizon, as well as the optimal simulation length that varies based on the specific dynamical tasks to be performed. Further improvements can be made in this regard, such as incorporating the length of the horizon as an optimized parameter to find the most suitable solution.
Overall, this research contributes to the understanding and application of model-based techniques in constrained trajectory optimization for non-rigid object manipulation. It provides insights into the importance of accurate models, appropriate cost functions, and the trade-offs involved in balancing simulation and control aspects.

Appendix A

Code

A.1 Rope_system.py

```
import jax
import jax.numpy as np
from jax import jit, jacfwd
from jax.config import config
config.update("jax_debug_nans", True)
jax.config.update('jax_enable_x64', True)
from functools import partial
from skspatial.objects import Sphere
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import animation
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d
matplotlib.rc('animation', html='jshtml')
import mpl_toolkits.mplot3d.art3d as art3d
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

class Arrow3D(FancyArrowPatch):
    # Arrow plotting code from:
    # https://stackoverflow.com/questions/22867620/putting-arrowheads-on-vectors-in-matplotlibs-3d-plot
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
```

```
28          self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
29          FancyArrowPatch.draw(self, renderer)
30
31  class DynamicalSystem:
32      def __init__(self, state_size, control_size):
33          self.state_size = state_size
34          self.control_size = control_size
35
36      def set_cost(self, Q, R):
37          # one step cost = x.T * Q * x + u.T * R * u
38          self.Q = Q
39          self.R = R
40
41      def get_cost(self):
42          return self.Q, self.R
43
44      def set_final_cost(self, Q_f):
45          self.Q_f=Q_f
46
47      def calculate_cost(self, x, u):
48          x=np.ravel(x)
49         # return 0.5*(x.T.dot(self.Q).dot(x)+u.T.dot(self.R).dot(u))
50
51          return 0.5*((x-self.goal).T.dot(self.Q).dot(x-self.goal)+u.T.
    dot(self.R).dot(u))
52
53      def calculate_final_cost(self,x):
54          x=np.ravel(x)
55          return 0.5*(x-self.goal).T.dot(self.Q_f).dot(x-self.goal)
56
57      def set_goal(self, x_goal):
58          self.goal = x_goal
59
60
61  class Rope(DynamicalSystem):
62      def __init__(self,n_masses,dt):
63          super().__init__(n_masses*6,3)
64          self.dt = dt
65          self.goal = np.zeros(self.state_size)
66          self.n_masses=n_masses
67          self.l_rest=1
68          self.m=0.3/self.n_masses
69          self.k_elastic=140.0
70          self.k_shear = 5.0
71          self.k_bend = 1
72          self.c_elastic = self.k_elastic / self.n_masses
73          self.c_shear = self.k_shear / self.n_masses
74          self.c_bend = self.k_bend / self.n_masses
75          g = 9.80665          # m/s^2
```

```
76          self.ag = np.array([0.0, 0.0, -g])  # acceleration due to
    gravity
77          self.actuated_masses = [0]
78
79      def init_position(self,xa,ya,za,i):
80          xb = xa
81          yb = ya
82          zb = za-self.l_rest*i
83          k=self.n_masses-za-1
84          if zb < 0:
85              zb=0
86              xb=xa+self.l_rest*(i-(self.n_masses-1)+k)
87          return np.array([xb, yb, zb], dtype=np.float32)
88
89      def initial_position(self,x,y,z):
90          p0 = np.array([self.init_position(x,y,z,i) for i in np.arange
    (self.n_masses)])
91          p0=p0.reshape(self.n_masses*3,1)
92          v0 = np.zeros((self.n_masses*3, 1))
93          s0 = np.vstack((p0, v0))
94          return s0
95
96      def goal_pos(self,xa,ya,za,i):
97          xb = xa+self.l_rest*i
98          yb = ya
99          zb = za
100         return np.array([xb, yb, zb], dtype=np.float32)
101
102     def goal_position(self,x,y,z):
103         p0 = np.array([self.goal_pos(x,y,z,i) for i in np.arange(self
    .n_masses)])
104         p0=p0.reshape(self.n_masses*3,1)
105         v0 = np.zeros((self.n_masses*3, 1))
106         s0 = np.vstack((p0, v0))
107         return s0
108
109     def return_position_actuated_mass(self,s):
110         p,_=np.vsplit(s,2)
111         p=p.reshape(self.n_masses,3)
112         return p[0,:]
113
114     def return_position_last2mass(self,s):
115         p,_=np.vsplit(s,2)
116         p=p.reshape(self.n_masses,3)
117         return p[self.n_masses-2:self.n_masses,:]
118
119     #@jit
120     def hooke_damped(self,p,v):
121         f = np.zeros_like(p)
```

```
122         for i in range(0, self.n_masses-1):
123             # Compute the displacement and distance between
    neighboring particles
124             delta_pos = p[i+1,:] - p[i,:]
125             delta_vel = v[i+1,:] - v[i,:]
126             dist = np.linalg.norm(delta_pos)
127 
128             dist_non_zero=np.maximum(dist, self.l_rest/10)
129 
130             u= delta_pos / dist_non_zero
131             x=(dist - self.l_rest) * u
132             x_dot=np.dot(delta_vel,u)*u
133             # Compute the force exerted by the spring
134             f_spring = self.k_elastic * x + self.c_elastic*x_dot +
    self.k_shear * x +  self.c_shear*x_dot
135             f=f.at[i].add(f_spring)
136             f=f.at[i+1].add(-f_spring)
137 
138             if i < self.n_masses-2:
139                 delta_pos = p[i+2,:] - p[i,:]
140                 delta_vel = v[i+2,:] - v[i,:]
141                 dist = np.linalg.norm(delta_pos)
142 
143                 dist_non_zero=np.maximum(dist, 2*self.l_rest/10)
144 
145                 u= delta_pos / dist_non_zero
146                 x=(dist - 2*self.l_rest) * u
147                 x_dot=np.dot(delta_vel,u)*u
148                 f_bend = self.k_bend * x + self.c_bend*x_dot
149                 f=f.at[i].add(f_bend)
150                 f=f.at[i+2].add(-f_bend)
151 
152 
153         return f
154 
155     @partial(jit, static_argnums=(0,))
156     def change_of_state(self,s, F_act):
157         s=s.reshape(6*self.n_masses,1)
158         p,v=np.vsplit(s,2)
159         p=p.reshape(self.n_masses,3)
160 
161         v=v.reshape(self.n_masses,3)
162         F = self.hooke_damped(p, v)
163         F = F.at[np.index_exp[self.actuated_masses, :]].add(F_act)
164 
165         a = F / self.m
166         a += self.ag
167         a=a.reshape(self.n_masses*3,1)
168         v=v.reshape(self.n_masses*3,1)
```

```
169         return np.vstack((v, a))
170
171     @partial(jit, static_argnums=(0,1,))
172     def RK4(self, f, x, u):
173         k1 = f(x, u)
174         k2 = f(x + k1 * (self.dt / 2),u)
175         k3 = f(x + k2 * (self.dt / 2),u)
176         k4 = f(x + k3 * self.dt,u)
177         x_new = x + (k1 + 2 * k2 + 2 * k3 + k4) * (self.dt / 6)
178         return x_new
179
180     def transition(self,x,u):
181         x=x.reshape(6*self.n_masses,1)
182         f=self.change_of_state
183         x_new=self.RK4(f,x,u)
184         p,v=np.vsplit(x_new,2)
185         p=p.reshape(self.n_masses,3)
186         v=v.reshape(self.n_masses,3)
187         #floor
188         z = p[:, 2]
189         jumpv=np.where(z <= 0, 0, v[:,2])
190         jumpp=np.where(z <= 0, 0, z)
191         v=v.at[:,2].set(jumpv)
192         p=p.at[:,2].set(jumpp)
193         #box
194         for k in range(self.n_masses):
195             if p[k,0] >= 3.3 and p[k,0] <=6.7:
196                 if p[k,1] >= -1.2 and p[k,1] <=1.2:
197                     if p[k,2] >= 1.8 and p[k,2] <=3.4:
198                         if np.isclose(p[k,2],2,atol=1e-1): #z axis
199                             v=v.at[k,2].set(-0.8*v[k,2])
200                             p=p.at[k,2].set(1.7)
201                         elif np.isclose(p[k,2],3.2,atol=1e-1):
202                             v=v.at[k,2].set(0.0)
203                             p=p.at[k,2].set(3.2)
204                         if np.isclose(p[k,0],3.5,atol=1e-1): #x axis
205                             v=v.at[k,0].set(-0.8*v[k,0])
206                             p=p.at[k,0].set(3.2)
207                         elif np.isclose(p[k,0],6.5,atol=1e-1):
208                             v=v.at[k,0].set(-0.8*v[k,0])
209                             p=p.at[k,0].set(6.8)
210                         if np.isclose(p[k,1],-1,atol=1e-1): #y axis
211                             v=v.at[k,1].set(-0.8*v[k,1])
212                             p=p.at[k,1].set(-1.3)
213                         elif np.isclose(p[k,1],1,atol=1e-1):
214                             v=v.at[k,1].set(-0.8*v[k,1])
215                             p=p.at[k,1].set(1.3)
216                 # elif np.isclose(p[k,2],4.2,atol=1e-1):
217                 #     v=v.at[k,2].set(0.0)
```

```
218                 #       p=p.at[k,2].set(4.3)
219         p=p.reshape(self.n_masses*3,1)
220         v=v.reshape(self.n_masses*3,1)
221         x_new=np.vstack((p,v))
222         return x_new
223 
224     #@partial(jit, static_argnums=(0,1,))
225     def transition_J(self,x,u):
226 
227         x_temp=x.reshape(6*self.n_masses,1)
228 
229         dynamics_jac_state = jacfwd(self.transition, argnums=0)(
    x_temp,u)
230 
231         dynamics_jac_control = jacfwd(self.transition, argnums=1)(
    x_temp,u)
232         A = dynamics_jac_state[:,1,:,1]
233         B = dynamics_jac_control[:,1,:]
234         # print(x)
235         return A,B
236 
237     def plot_rope(self, ax, s,
238                   xlim=[-2, 5],
239                   ylim=[-2, 3],
240                   zlim=[0, 5]):
241         ux=6.5
242         uy=1
243         uz=3.2
244         lx=3.5
245         ly=-1
246         lz=2.0
247 
248         p,_=np.vsplit(s,2)
249         p=p.reshape(self.n_masses,3)
250         x, y, z = np.transpose(p)
251         ax.clear()  # necessary for the animations
252         ax.set_xlim(xlim)
253         ax.set_ylim(ylim)
254         ax.set_zlim(zlim)
255 
256         ax.scatter(x, y, z, c='red', s=10)
257         ax.plot(x,y,z,c='blue')
258         x = [lx,ux,ux,lx],[lx,ux,ux,lx],[lx,lx,lx,lx],[lx,lx,ux,ux],[
    lx,lx,ux,ux],[ux,ux,ux,ux]
259         y = [ly,ly,uy,uy],[ly,ly,uy,uy],[ly,ly,uy,uy],[uy,uy,uy,uy],[
    uy,uy,uy,uy],[ly,ly,uy,uy]
260         z = [lz,lz,lz,lz],[uz,uz,uz,uz],[lz,uz,uz,lz],[lz,uz,uz,lz],[
    lz,uz,uz,lz],[lz,uz,uz,lz]
261 
```

```
262         surfaces = []
263 
264         for i in range(len(x)):
265             surfaces.append( [list(zip(x[i],y[i],z[i]))] )
266 
267         for surface in surfaces:
268             ax.add_collection3d(Poly3DCollection(surface, facecolors=
    'cyan', linewidths=1, edgecolors='b', alpha=.2))
269 
270         sphere = Sphere([0,0,4],3)
271         sphere.plot_3d(ax, alpha=0.2)
272         ax.set_xlabel('x')
273         ax.set_ylabel('y')
274         ax.set_zlabel('z')
275 
276 
277     def plot_arrow(self,ax, start, force, scale):
278         end = start + scale * force
279         x0, y0, z0 = start
280         x1, y1, z1 = end
281         a = Arrow3D([x0, x1], [y0, y1], [z0, z1], mutation_scale=10,
    lw=3, arrowstyle="-|>", color="mediumseagreen", zorder=10)
282         ax.add_artist(a)
283 
284     def animate_cloth(self, horizon, s_history, dt, fps=60, F_history
    =None, force_scale=0.2, gifname=None):
285         fig = plt.figure(figsize=(5, 5), dpi=100)
286         fig.subplots_adjust(0,0,1,1,0,0)
287         ax = fig.add_subplot(111, projection='3d')
288         plt.close()  # prevents duplicate output
289 
290         fps_simulation = 1 / dt
291         skip = np.floor(fps_simulation / fps).astype(np.int32)
292         fps_adjusted = fps_simulation / skip
293         print('fps was adjusted to:', fps_adjusted)
294         horizon=horizon-1
295 
296         def animate(i):
297             j = min(i * skip, horizon)
298 
299             p = s_history[:,j].reshape(self.n_masses*6,1)
300 
301             self.plot_rope(ax, p)
302             if F_history is not None:
303                 self.plot_arrow(ax, np.ravel(p[0:3]), np.ravel(
    F_history[:,j]), force_scale)
304 
305 
```

```
306         n_frames = (horizon) // skip + 1  # this +1 is for the
    initial frame
307         if not (horizon) % skip == 0:
308             n_frames += 1  # this +1 is to ensure the final frame is
    shown
309 
310         anim = animation.FuncAnimation(fig, animate, frames=n_frames,
     interval=1000*dt*skip)
311 
312         if gifname is not None:
313             anim.save(gifname + '.gif', writer='imagemagick')
314 
315         return anim
316 
317     def draw_trajectories(self,ax, x_trajectories,p0,pend):
318         #self.plot_rope(ax,p0)
319         self.plot_rope(ax,pend)
320         plt.plot(x_trajectories[0,:], x_trajectories[1, :],
    x_trajectories[2,:],color='r')
321         ax.set_aspect("auto")
322         plt.grid()
323         plt.show()
```

A.2 AL_iLQR.py

```
1 from iLQR import LQR
2 import numpy as np
3 import matplotlib.pyplot as plt
4 
5 class AL_iLQR:
6     def __init__(self, system, initial_state, horizon):
7         self.system=system
8         self.horizon=horizon
9         self.x_traj=initial_state @ np.ones((1,self.horizon + 1))
10        self.u_traj=0.001* np.ones((self.system.control_size,self.
    horizon))
11        self.initial_state=np.copy(initial_state)
12        self.constraints = []
13        self.tol = 0.01
14        self.tol_c = 0.5
15        self.penalty=10 #phi
16 
17        self.max_iterations=100
18        self.max_iterations_al=0
19 
```

```
20      def add_constraint(self, constraint):
21          self.constraints.append(constraint)
22
23      def algorithm(self):
24          # Initialization of lambda, mu and phi
25          self.multipliers=np.zeros((len(self.constraints),self.horizon
    ))
26          self.mu=0.1*np.ones((len(self.constraints),self.horizon))
27          iter_al=0
28
29          # initial value of the constraints
30          c=np.ones((len(self.constraints),self.horizon))
31          for i in range(len(self.constraints)): # for each constraint
32              c[i,0]=self.constraints[i].evaluate_constraint(self.
    initial_state)
33
34          lqr=LQR(self.system, self.initial_state, self.horizon, self.
    constraints)
35          while np.max(c) > self.tol_c and iter_al<=self.
    max_iterations_al:
36              #minimize Lagrangian using iLQR
37              iter_lq=0
38              J_new=0
39              J_prev=0
40              # compute J with X and U
41              for i in range(self.horizon):
42                  J_new += self.system.calculate_cost(self.x_traj[:,i],
    self.u_traj[:,i])
43              J_new += self.system.calculate_final_cost(self.x_traj[:,
    self.horizon])
44
45              x_new=np.copy(self.x_traj)
46              u_new=np.copy(self.u_traj)
47              while abs(J_new-J_prev) > self.tol and iter_lq<=self.
    max_iterations:
48                  J_prev=J_new
49                  lqr.backward_pass(self.multipliers,self.mu,x_new,
    u_new)
50                  x_new,u_new,J_new=lqr.forward_pass(x_new,u_new,J_prev
    )
51                  iter_lq +=1
52                  print("Difference between the previou and the new
    costs: ",abs(J_new-J_prev))
53                  print("N of iteration of iLQR: ",iter_lq)
54              print("Number of required iterations for iLQR: ",iter_lq)
55              self.x_traj=np.copy(x_new)
56              self.u_traj=np.copy(u_new)
57              final_cost=self.system.calculate_final_cost(self.x_traj
    [:,self.horizon])
```

```
58              print(f"Final cost for horizon {self.horizon} = {
    final_cost}")
59              #update value for multipliers
60              for i in range(len(self.constraints)): # for each
    constraint
61                  for k in range(self.horizon):
62
63                      c[i,k]=self.constraints[i].evaluate_constraint(
    x_new[:,k]) # constraint i at step k
64
65                      self.multipliers[i,k]=max(0, self.multipliers[i,k
    ] + self.mu[i,k]*c[i,k])
66                      self.mu[i,k]=self.penalty*self.mu[i,k]
67              print(self.multipliers)
68              iter_al=iter_al+1
69
70              print("Maximum violation of the constraints: ",np.max(c))
71
72              self.system.animate_cloth(self.horizon,self.x_traj,self.
    system.dt,F_history=self.u_traj, gifname=str(iter_al-1)+"_"+str(
    self.system.dt)+"_"+str(J_new)+"_"+str(np.max(c)))
73              print(np.max(self.u_traj))
74              print(np.min(self.u_traj))
75          print("Maximum violation of the constraints: ",np.max(c))
76          print("Number of required iterations for AL-iLQR: ",iter_al)
```

A.3 LQR.py

```
1  import numpy as np
2  from numpy.linalg import inv
3  import shelve
4
5  class LQR:
6      def __init__(self, system, initial_state, horizon, constraints):
7          self.system=system
8          self.horizon=horizon
9          self.initial_state = np.copy(initial_state)
10         self.reg_factor_u = 1e-3
11         self.constraints=constraints[:]
12         self.multipliers=np.zeros((len(self.constraints),self.horizon
    ))
13         self.mu=np.zeros((len(self.constraints),self.horizon))
14
15         self.penalty=10
16         self.alpha=1
```

```
17          self.e_constraint=0.5
18          self.d=np.zeros((self.system.control_size, self.horizon))
19          self.K=np.zeros((self.system.control_size, self.system.
    state_size, self.horizon))
20          self.delta_V1=np.zeros(self.horizon)
21          self.delta_V2=np.zeros(self.horizon)
22          self.max_iter=100
23          self.fs=10
24
25      def set_initial_trajectories(self, x_traj, u_traj):
26          self.x_traj = np.copy(x_traj)
27          self.u_traj = np.copy(u_traj)
28
29      def forward_pass(self,x_up,u_up,J_prev):
30          done=0
31          new_J=0
32          current_J=J_prev
33          self.alpha=1
34          x_new_traj = np.zeros((self.system.state_size, self.horizon +
     1))
35          u_new_traj = np.zeros((self.system.control_size, self.horizon
    ))
36          x = np.copy(self.initial_state)
37          iter=0
38          while done==0:
39              for i in range(0,self.horizon,self.fs):
40                  x_new_traj[:, i] = np.copy(np.ravel(x))
41                  delta_x = x - x_up[:, i].reshape(6*self.system.
    n_masses,1)
42                  u=u_up[:,i]+np.ravel(self.K[:,:,i].dot(delta_x))+self
    .alpha*self.d[:,i]
43                  # +
44                  u_new_traj[:, i] = np.copy(np.ravel(u))
45                  new_J += self.system.calculate_cost(x, u)
46                  x = self.system.transition(x, u)
47                  for j in range(1,self.fs):    # run the simulation
    for 10 times
48                      u_new_traj[:, i+j] = np.copy(np.ravel(u))
49                      x_new_traj[:, i+j] = np.copy(np.ravel(x))
50                      new_J += self.system.calculate_cost(x, u)
51                      x = self.system.transition(x, u)
52                      x=x+np.random.uniform(low=-5e-3,high=5e-3,size
    =(30,1))
53
54              x_new_traj[:, self.horizon] = np.copy(np.ravel(x))
55              new_J += self.system.calculate_final_cost(x)
56              if new_J > 1e5:
57                  iter+=1
58                  if iter == self.max_iter:
```

```
59                      print('Max number of iteration for forward pass')
60                      return x_up,u_up, J_prev
61                  new_J=0
62                  self.alpha=0.8*self.alpha
63                  x_new_traj = np.zeros((self.system.state_size, self.
    horizon + 1))
64                  u_new_traj = np.zeros((self.system.control_size, self
    .horizon))
65                  x = np.copy(self.initial_state)
66              else:
67                  adelta_V=self.alpha*self.delta_V1+self.alpha**2*self.
    delta_V2
68                  J=0
69                  for i in range(self.horizon):
70                      J+= self.system.calculate_cost(x_up[:,i], u_up[:,
    i])
71                  J+= self.system.calculate_final_cost(x_up[:,self.
    horizon])
72
73                  z=(J-new_J)/(-np.sum(adelta_V))
74                  print("z value for the line search: ",z)
75                  print("Cost of the previous trajectory ",J)
76                  print("Cost of the new trajectory ",new_J)
77                  iter+=1
78                  print("N  of iteration: ",iter)
79                  if iter == self.max_iter:
80                      print('Max number of iteration for forward pass')
81                      return x_up,u_up, J_prev
82                  if (z < 15 and z > 1e-8):
83                      x_up=np.copy(x_new_traj)
84                      u_up=np.copy(u_new_traj)
85                      print("Cost for the new optimal trajectory",new_J
    )
86                      done=1
87                      print("Forward pass required: ", iter, "
    iterations")
88                      iter=0
89                      return x_up,u_up,new_J
90                  elif abs(new_J-J) < 0.005:
91                      print("Cost for the new optimal trajectory",J)
92                      done=1
93                      print("Forward pass required: ", iter, "
    iterations")
94                      iter=0
95                      return x_up,u_up,J
96                  else:
97                      new_J=0
98                      self.alpha=0.8*self.alpha
```

```
99                      x_new_traj = np.zeros((self.system.state_size,
    self.horizon + 1))
100                     u_new_traj = np.zeros((self.system.control_size,
    self.horizon))
101                     x = np.copy(self.initial_state)
102
103
104     def backward_pass(self, lam, mu, x_new,u_new):
105         #print("Backward pass")
106         self.mu=np.copy(mu)
107         self.multipliers=np.copy(lam)
108
109         # definition of pn and Pn
110         ln_xx = np.copy(self.system.Q_f)
111         ln_x = self.system.Q_f @ (x_new[:, self.horizon] - self.
    system.goal)
112         self.Iu=self.mu[:,self.horizon-1]*np.diag(np.ones(len(self.
    constraints)))
113         C=np.ones(len(self.constraints))
114         C_x=np.zeros((len(self.constraints),self.system.state_size))
115         for i in range(len(self.constraints)):
116             C[i]=self.constraints[i].evaluate_constraint(x_new[:,
    self.horizon])
117
118         # checking if the constraint is active
119             if C[i] <- self.e_constraint and self.multipliers[i,
    self.horizon-1] == 0:
120                 self.Iu[i,i] = 0 # it means that the constraint is
    not active
121
122             C_x[i,:]=self.constraints[i].evaluate_constraint_J(x_new
    [:, self.horizon])
123         cu=np.zeros((len(self.constraints),self.system.control_size))
     # no constraints on control trajectory
124         pn=ln_x+C_x.T @ (self.multipliers[:,self.horizon-1]+self.Iu @
     C)
125         Pn=ln_xx+C_x.T @ self.Iu @ C_x
126
127         for i in range(self.horizon - 1, -1, -self.fs):
128             print(i)
129             u = u_new[:, i]
130             x = x_new[:, i]
131             # definition of derivatives of the cost function
132             l_xt = self.system.Q @ (x - self.system.goal)
133             l_ut = self.system.R @ u
134             l_uxt = np.zeros((self.system.control_size, self.system.
    state_size))
135             l_xxt = np.copy(self.system.Q)
136             l_uut = np.copy(self.system.R)
```

```
137             self.Iu=self.mu[:,i]*np.diag(np.ones(len(self.constraints
    )))
138             for j in range(len(self.constraints)):
139                 C_x[j,:]=self.constraints[j].evaluate_constraint_J(x)
140                 C[j]=self.constraints[j].evaluate_constraint(x)
141                 if C[j] <- self.e_constraint and self.multipliers[j
    ,i] == 0:
142                     self.Iu[j,j] = 0
143             if i+1==self.horizon:
144                 p=pn
145                 P=Pn
146             else:
147                 p=Q_x + self.K[:,:,i+1].T @ Q_uu @ self.d[:,i+1] +
    self.K[:,:,i+1].T @ Q_u + Q_ux.T @ self.d[:,i+1]
148                 P=Q_xx + self.K[:,:,i+1].T @ Q_uu @ self.K[:,:,i+1]+
    self.K[:,:,i+1].T @ Q_ux + Q_ux.T @ self.K[:,:,i+1]
149
150             A, B = self.system.transition_J(x,u)  #matrices A and B
    from dynamics
151             Q_x = l_xt + A.T @ p + C_x.T @ (self.multipliers[:,i]+
    self.Iu @ C)
152             Q_u = l_ut + B.T @ p + cu.T @ (self.multipliers[:,i]+self
    .Iu @ C)
153             Q_ux = l_uxt + B.T @ P @ A + cu.T @ self.Iu @ C_x
154             Q_uu = l_uut + B.T @ P @ B + cu.T @ self.Iu @ cu + self.
    reg_factor_u * np.identity(self.system.control_size)
155             Q_xx = l_xxt + A.T @ P @ A + C_x.T @ self.Iu @ C_x
156
157             if np.all(np.linalg.eigvals(Q_uu) > 0):
158                 for j in range(0,self.fs):
159                     self.K[:,:,i-j]=-inv(Q_uu) @ Q_ux
160                     self.d[:,i-j]=-inv(Q_uu) @ Q_u
161                     self.delta_V1[i-j]=self.d[:,i-j].T @ Q_u
162                     self.delta_V2[i-j]=0.5*self.d[:,i-j].T @ Q_uu @
    self.d[:,i-j]
163             else:
164                 self.reg_factor_u = self.reg_factor_u*5
```

A.4 Constraints.py

```
1 import numpy as np
2
3 class SphereConstraint:
4     def __init__(self, center, r, system):
5         self.center = center
```

```
6           self.r = r
7           self.system = system
8
9       def evaluate_constraint(self, x):
10          #evaluate the state to see if the constraint is violated
11          x=x.reshape(6*self.system.n_masses,1)
12          x_next = self.system.transition(x, np.zeros(self.system.
    control_size))
13
14          length = (x_next[0] - self.center[0])**2 + (x_next[1] - self.
    center[1])**2 + (x_next[2] - self.center[2])**2
15          return -(self.r**2 - length)
16
17      def evaluate_constraint_J(self, x):
18          #evolve the system for one to evaluate constraint
19          x=x.reshape(6*self.system.n_masses,1)
20          x_next = self.system.transition(x, np.zeros(self.system.
    control_size))
21          result = np.zeros(x.shape)
22          result[0] = 2*(x_next[0] - self.center[0])
23          result[1] = 2*(x_next[1] - self.center[1])
24          result[2] = 2*(x_next[2] - self.center[2])
25          result[self.system.n_masses] = 2*(x_next[0] - self.center[0])
     * self.system.dt
26          result[self.system.n_masses+1] = 2*(x_next[1] - self.center
    [1]) * self.system.dt
27          result[self.system.n_masses+2] = 2*(x_next[2] - self.center
    [2]) * self.system.dt
28          return np.ravel(result)
29
30
31  class BoxConstraint:
32      def __init__(self,ux,lx,uy,ly,uz,lz,system, horizon):
33          self.ux=ux
34          self.uy=uy
35          self.uz=uz
36          self.lx=lx
37          self.ly=ly
38          self.lz=lz
39          self.system=system
40          self.horizon=horizon
41
42      def evaluate_constraint(self,x):
43          x=x.reshape(6*self.system.n_masses,1)
44          x_next = self.system.transition(x, np.zeros(self.system.
    control_size))
45          p,v=np.vsplit(x_next,2)
46          p=p.reshape(self.system.n_masses,3)
47          v=v.reshape(self.system.n_masses,3)
```

```
48          for i in range(self.system.n_masses):
49              if p[i,0] >= self.lx and p[i,0] <=self.ux:
50                  if p[i,1] >= self.ly and p[i,1] <=self.uy:
51                      if p[i,2] >= self.lz and p[i,2] <=self.uz:
52                          return 1
53          return -1
54
55
56
57      def evaluate_constraint_J(self, x):
58          x=x.reshape(6*self.system.n_masses,1)
59
60          result = -np.ones(x.shape) * self.system.dt
61          for i in range(self.system.n_masses*3):
62              result[i]= -1
63          return np.ravel(result)
```

A.5 test.py

```
1  from rope_system import Rope
2  import numpy as np
3  from AL_iLQR import AL_iLQR
4  #import shelve
5  from constraints import SphereConstraint,BoxConstraint
6  import matplotlib.pyplot as plt
7  if __name__ == '__main__':
8      # Initializing the object
9      system=Rope(5,0.001)
10     # Initial configuration
11     x0=system.initial_position(0,0,4)
12     # Set running cost matrices
13     fig = plt.figure(figsize=(8, 6), dpi=100)
14     ax = fig.add_subplot(111, projection='3d')
15     system.plot_rope(ax,x0)
16     plt.savefig("traj.png")
17
18     Q=1e-4*np.identity(system.state_size)
19     for i in range(3*system.n_masses):
20         Q[i, i] = 1e-4 # to change the weight for the position
21     system.set_cost(Q, 0.005*np.identity(system.control_size))
22     Q_f = 1e-4*np.identity(system.state_size)
23     # Set final cost matrix
24     for i in range(3*system.n_masses):
25         if i >= (3*system.n_masses -9):
```

```
            Q_f[i, i] = 150 # the last three masses have more relevance in minimizing the cost function
    system.set_final_cost(Q_f)
    # Set the final goal
    height=4.2
    system.set_goal(np.ravel(system.goal_position(1,0,height)))
    print(f"The height for the final target is {height}")
    horizon=2000
    name_to_save=5.03e-3
    # definition of the constraints
    constraint1=SphereConstraint(system.return_position_actuated_mass(x0), 3, system)
    constraint2=BoxConstraint(6.6,3.4,1.1,-1.1,3.3,1.7,system,horizon)
    # starting the solver
    solver=AL_iLQR(system,x0, horizon)
    solver.add_constraint(constraint1)
    solver.add_constraint(constraint2)
    # solver.system.animate_cloth(horizon, solver.x_traj, system.dt, F_history=solver.u_traj, gifname="prova2")

    solver.algorithm()
    print(f"The maximum and minimum values for the control input are: {np.max(solver.u_traj)} and {np.min(solver.u_traj)}")
    print(f"The maximum and minimum values for the velocities are: {np.max(solver.x_traj[15:30,:])} and {np.min(solver.x_traj[15:30,:])}")
    #fig = plt.figure(figsize=(8, 6), dpi=100)
    fig = plt.figure(figsize=(8, 6), dpi=100)
    ax = fig.add_subplot(111, projection='3d')
    solver.system.draw_trajectories(ax,solver.x_traj,solver.x_traj[:,0],solver.x_traj[:,horizon])
    plt.savefig("trajnoise"+str(name_to_save)+"def.png")
```

Bibliography

[1] P. Jiménez. «Survey on model-based manipulation planning of deformable objects». In: *Robotics and Computer-Integrated Manufacturing* 28.2 (2012), pp. 154–163. ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2011.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0736584511000986 (cit. on p. 5).

[2] Félix Nadon, Angel J. Valencia, and Pierre Payeur. «Multi-Modal Sensing and Robotic Manipulation of Non-Rigid Objects: A Survey». In: *Robotics* 7.4 (2018). ISSN: 2218-6581. DOI: 10.3390/robotics7040074. URL: https://www.mdpi.com/2218-6581/7/4/74 (cit. on pp. 5, 12).

[3] Taylor A. Howell, Brian E. Jackson, and Zachary Manchester. «ALTRO: A Fast Solver for Constrained Trajectory Optimization». In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 7674–7679. ISBN: 1728140048 (cit. on p. 6).

[4] D. H. Jacobson and D.Q. Mayne. *Differential dynamic programming*. Modern analytic and computational methods in science and mathematics ; 24. New York, 1970 (cit. on p. 6).

[5] Markus Giftthaler and Jonas Buchli. «A projection approach to equality constrained iterative linear quadratic optimal control». In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. Ithaca: IEEE, 2018, pp. 61–66. ISBN: 9781538646786 (cit. on p. 7).

[6] L.-Z. Liao and C.A. Shoemaker. «Convergence in unconstrained discrete-time differential dynamic programming». In: *IEEE transactions on automatic control* 36.6 (1991), pp. 692–706. ISSN: 0018-9286 (cit. on p. 7).

[7] Daniel M. Murray and Sidney J. Yakowitz. «Differential dynamic programming and Newton's method for discrete optimal control problems». In: *Journal of Optimization Theory and Applications* 43 (1984), pp. 395–414. DOI: https://doi.org/10.1007/BF00934463 (cit. on p. 7).

[8] Daniel M. Murray and Sidney J. Yakowitz. «Constrained differential dynamic programming and its application to multireservoir control». In: *Water Resources Research* 15.5 (1979), pp. 1017–1027. DOI: https://doi.org/10.1029/WR015i005p01017. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/WR015i005p01017. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/WR015i005p01017 (cit. on p. 7).

[9] S. Yakowitz. «The stagewise Kuhn-Tucker condition and differential dynamic programming». In: *IEEE transactions on automatic control* 31.1 (1986), pp. 25–30. ISSN: 0018-9286 (cit. on pp. 7, 17).

[10] Yuval Tassa, Nicolas Mansard, and Emo Todorov. «Control-limited differential dynamic programming». In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 1168–1175. DOI: 10.1109/ICRA.2014.6907001 (cit. on p. 7).

[11] Yuichiro Aoyama, George Boutselis, Akash Patel, and Evangelos A Theodorou. «Constrained Differential Dynamic Programming Revisited». In: *arXiv.org* (2020). ISSN: 2331-8422 (cit. on pp. 7, 17).

[12] T. C. Lin and J. S. Arora. «Differential dynamic programming technique for constrained optimal control». In: *Computational Mechanics* 9.1 (1991), pp. 41–53. ISSN: 1432-0924. DOI: 10.1007/BF00369914. URL: https://doi.org/10.1007/BF00369914 (cit. on pp. 7, 17).

[13] Zhaoming Xie, C. Karen Liu, and Kris Hauser. «Differential dynamic programming with nonlinear constraints». In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 695–702. DOI: 10.1109/ICRA.2017.7989086 (cit. on pp. 7, 17).

[14] Weiwei Li and Emanuel Todorov. «Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems.» In: *Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics, (ICINCO 2004)* 1 (Jan. 2004), pp. 222–229 (cit. on p. 8).

[15] Simon Zimmermann, Roi Poranne, and Stelian Coros. «Dynamic Manipulation of Deformable Objects With Implicit Integration». In: *IEEE robotics and automation letters* 6.2 (2021), pp. 4209–4216. ISSN: 2377-3766 (cit. on p. 10).

[16] J. Montagnat, H. Delingette, and N. Ayache. «A review of deformable surfaces: topology, geometry and deformation». In: *Image and vision computing* 19.14 (2001), pp. 1023–1040. ISSN: 0262-8856 (cit. on p. 11).

[17] Mathieu Salzmann and Pascal Fua. «Linear Local Models for Monocular Reconstruction of Deformable Surfaces». In: *IEEE transactions on pattern analysis and machine intelligence* 33.5 (2011), pp. 931–944. ISSN: 0162-8828 (cit. on p. 11).

[18] Fouad F. Khalil and Pierre Payeur. «Dexterous Robotic Manipulation of Deformable Objects with Multi-Sensory Feedback - a Review». In: *Robot Manipulators*. Ed. by Agustin Jimenez and Basil M Al Hadithi. Rijeka: IntechOpen, 2010. Chap. 28. DOI: 10.5772/9183. URL: https://doi.org/10.5772/9183 (cit. on p. 12).

[19] Hang Yin, Anastasia Varava, and Danica Kragic. «Modeling, learning, perception, and control methods for deformable object manipulation». In: *Science robotics* 6.54 (2021). ISSN: 2470-9476 (cit. on pp. 12, 13).

[20] Jan Bender, Matthias Müller, Miguel A. Otaduy, Matthias Teschner, and Miles Macklin. «A Survey on Position-Based Simulation Methods in Computer Graphics». In: *Computer graphics forum* 33.6 (2014), pp. 228–251. ISSN: 0167-7055 (cit. on p. 12).

[21] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. «XPBD: position-based simulation of compliant constrained dynamics». In: *Proceedings of the 9th International Conference on motion in games*. MIG '16. ACM, 2016, pp. 49–54. ISBN: 1450345921 (cit. on p. 12).

[22] Fei Liu, Entong Su, Jingpei Lu, Mingen Li, and Michael C. Yip. «Robotic Manipulation of Deformable Rope-Like Objects Using Differentiable Compliant Position-Based Dynamics». In: *IEEE robotics and automation letters* 8.7 (2023), pp. 3964–3971. ISSN: 2377-3766 (cit. on pp. 12, 79).

[23] Veronica E. Arriola-Rios and Jeremy L. Wyatt. «A Multimodal Model of Object Deformation Under Robotic Pushing». In: *IEEE transactions on cognitive and developmental systems* 9.2 (2017), pp. 153–169. ISSN: 2379-8920 (cit. on p. 13).

[24] Puren Guler, Alessandro Pieropan, Masatoshi Ishikawa, and Danica Kragic. «Estimating deformability of objects using meshless shape matching». In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 5941–5948. ISBN: 9781538626825 (cit. on p. 13).

[25] Cheng Chi, Benjamin Burchfiel, Eric Cousineau, Siyuan Feng, and Shuran Song. «Iterative Residual Policy for Goal-Conditioned Dynamic Manipulation of Deformable Objects». In: *Proceedings of Robotics: Science and Systems (RSS)*. 2022 (cit. on p. 14).

[26] A.A. Nippun Kumaar and Sudarshan TSB. «Mobile Robot Programming by Demonstration». In: *2011 Fourth International Conference on Emerging Trends in Engineering Technology*. 2011, pp. 206–209. DOI: 10.1109/ICETET.2011.30 (cit. on p. 14).

[27] A.A. Nippun Kumaar and T.S.B. Sudarshan. «Learning from Demonstration with State Based Obstacle Avoidance for Mobile Service Robots». In: *Applied Mechanics and Materials* 394 (2013), pp. 448–455. ISSN: 1660-9336 (cit. on p. 14).

[28] Heni Ben Amor, David Vogt, Marco Ewerton, Erik Berger, Bernhard Jung, and Jan Peters. «Learning responsive robot behavior by imitation». In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 3257–3264. DOI: 10.1109/IROS.2013.6696819 (cit. on p. 14).

[29] Fabricio Oliveira. *Optimisation Notes*. 2018. URL: https://github.com/gamma-opt/optimisation-notes (cit. on pp. 15–17, 27).

[30] Glen Alleman. «Is There an Underlying Theory of Software Project Management? (A critique of the transformational and normative views of project management)». In: (Oct. 2002) (cit. on p. 19).

[31] Russ Tedrake. *Underactuated Robotics. Algorithms for Walking, Running, Swimming, Flying, and Manipulation*. 2023. URL: https://underactuated.csail.mit.edu (cit. on p. 22).

[32] Manohar Srikanth, P.C. Mathias, Vijay Natarajan, Prakash Naidu, and Tim Poston. «Visibility volumes for interactive path optimization». In: *Visual Computer* 24 (July 2008), pp. 635–647. DOI: 10.1007/s00371-008-0244-x (cit. on p. 34).

[33] Veronica E. Arriola-Rios, Puren Guler, Fanny Ficuciello, Danica Kragic, Bruno Siciliano, and Jeremy L. Wyatt. «Modeling of Deformable Objects for Robotic Manipulation: A Tutorial and Review». In: *Frontiers in Robotics and AI* 7 (2020). ISSN: 2296-9144. DOI: 10.3389/frobt.2020.00082. URL: https://www.frontiersin.org/articles/10.3389/frobt.2020.00082 (cit. on p. 34).

[34] Il-Kwon Jeong and Inho Lee. «An Oriented Particle and Generalized Spring Model for Fast Prototyping Deformable Objects». In: *Eurographics 2004 - Short Presentations*. Ed. by M. Alexa and E. Galin. Eurographics Association, 2004. DOI: 10.2312/egs.20041014 (cit. on p. 36).

[35] victorlouisdg. *differentiable-cloth-folding*. 2020. URL: https://github.com/Victorlouisdg/differentiable-cloth-folding (cit. on p. 36).