Politecnico di Torino

Master’s degree in Aerospace Engineering

A.Y. 2024/2025

Graduation Session October 2025

A digital twin of an air-bearing platform for tethered satellite systems: from tether deployment to post-deployment control

Supervisors:
Prof. Paolo Maggiore
Dr. Giuseppe Governale
Prof. Stephanie Lizy-Destrez

Candidate:
Edoardo De Blasi

Acknowledgements

I would like to express my sincere gratitude to all the people who made this work possible.

I want to thank Doctor Giuseppe Governale for his availability, guidance, and invaluable advice throughout all the stages of this project, as well as Professor Stéphanie Lizy-Destrez and Professor Paolo Maggiore for their help and supervision.

I wish to thank the Astradors team for their warm welcome, their guidance, and the many nice moments we shared together.

I am deeply grateful to my family for their constant support, and to my cousins: Stefania, who inspired my passion for science through the fascinating stories she told me as a child, and for her help during my high school years; Anna Maria and Lorenzo, for welcoming me with warmth and for the many pleasant moments we shared, which brightened my time in Turin.

My heartfelt thanks also go to my friend Stefano, who supported me during the most difficult moments here in Turin and helped me unwind when I needed it most: this achievement is yours too.

Finally, I would like to thank my friends from the Horus project and my Erasmus friends, for making me feel at home wherever I was. The laughs and beautiful evenings we shared are memories I will always keep in my heart.

Abstract

Tethered Satellite Systems (TSS) represent a promising technology for a wide range of space applications, including orbital maneuvering, de-orbiting operations, and in-orbit servicing. These systems enable novel mission concepts that can reduce propellant consumption and mission costs, while also supporting sustainable space operations. On the other hand, TSS are characterized by complex dynamics both during deployment and post-deployment phases, making their modeling and experimental validation extremely important.

Given the high costs associated with space missions, ground testing has always represented an important opportunity to reduce overall mission expenses. Nowadays, however, a new trend is the development of digital twins (namely digital models of experimental setups) which make it possible to further reduce costs and to carry out an unlimited number of tests. To address this challenge, this thesis presents the development of a digital twin of an air-bearing platform for tethered satellite systems, implemented in Unity™.

The first and most important step was the analysis of the existing literature on tethered satellite systems. This review aimed to identify, with the support of tools such as the Analytic Hierarchy Process (AHP), a TSS architecture capable of ensuring a safe deployment and maximizing the tether's expected lifetime in the harsh environment of Low Earth Orbit (LEO). To enhance the physical understanding of the system through an intuitive 3D representation, a digital twin was developed using a 3D modeling software. Among the tools examined, the software providing the optimal trade-off among key criteria such as physical fidelity and accessibility was selected. After the TSS architectural design was defined, attention was devoted to the problem of scaling the real system to the dimensions of the experimental setup. Subsequently, the implementation of the air-bearing platform in Unity™ was undertaken. Two scenarios were developed: one in which the tether is already deployed, and one concerning the deployment phase. In the first case, two position control algorithms (PID and LQR) and a PID controller for tether tension were designed and tested, while in the second case a control strategy was implemented to halt the deployment once the desired tether length had been reached, along with a PID tension control algorithm that was likewise developed and validated. It is important to note that the scenario with the tether already deployed was addressed first. This allowed for the immediate testing of the modeling approaches used in Unity™. Only after verifying their correctness was the deployment case developed. This case required more complex modeling due to the helical winding of the tether around the spool.

Finally, based on the results, it can be concluded that the objective of developing a digital twin of an air-bearing platform capable of simulating tether deployment, reproducing post-deployment dynamics, and validating control strategies has been successfully accomplished. Future developments may involve integrating sensors, incorporating force exchange and friction in the tether-spool interaction, and creating a more accurate model of the spool and floater geometries.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, tethered satellite systems have emerged as a promising solution to several pressing challenges in space exploration and orbital operations, ranging from debris mitigation and satellite deorbiting to formation flying and momentum management. Their potential to provide efficient, low-cost, and sustainable alternatives to conventional propulsion methods has stimulated interest within the aerospace community. However, the inherent complexity of tether dynamics, coupled with the difficulties of reproducing microgravity conditions on Earth, makes the experimental validation of such systems a formidable task. Direct testing in orbit remains prohibitively expensive and logistically demanding, which further amplifies the need for ground-based experimental platforms capable of offering realistic yet accessible environments.

Among the available non orbital reduced-gravity testing facilities, air-bearing platforms represent a particularly attractive option, due to their lower cost and greater accessibility compared to other types of testing, such as drop towers, sounding rockets and parabolic flights. By drastically reducing friction between the test bench and the floater, they enable the emulation of reduced-gravity conditions in a controlled laboratory environment. This makes them particularly suitable for investigating two-dimensional analogues of orbital maneuvers, such as rendezvous, docking, and tether deployment. In this way, critical tethered satellite missions aspects (from deployment mechanisms to guidance, navigation, and control algorithms) can be evaluated and refined before engaging in costly in-orbit demonstrations.

At the same time, the development of advanced software has unlocked new avenues for complementing physical experiments. In particular, digital twins (virtual representations of physical systems) are increasingly recognized as potent tools for design, testing, and operational support. Implementing a digital twin of a tethered satellite system offers several advantages: it minimizes reliance on costly hardware facilities, enables researchers to explore a diverse range of configurations and materials, and facilitates the testing of control laws under various scenarios, all without the inherent risks and expenses associated with space missions.

Modern 3D modeling software, such as Blender, Unity™, and NVIDIA IsaacSim, further enhances these opportunities. Unlike traditional simulation software, these platforms provide highly realistic visualization, advanced physics modeling, and the ability to integrate control algorithms through scripts. This makes them particularly well-suited to reproducing the nonlinear and contact-rich dynamics of tethers, such as collisions, oscillations, or spool interactions, which are otherwise challenging to capture. In this context, developing a digital twin of an air-bearing platform for tethered satellite systems allows researchers to bridge the gap between theoretical models, numerical simulations, and experimental validation. This ultimately supports the design of more robust and long-lived tethered missions in space.

1.1 Thesis outline

This thesis aims to guide the reader through the theoretical background, the development, and validation of a digital twin for tethered satellite systems.

The thesis starts with a review of the current literature. It gives context and highlights the main challenges in the field. Based on this foundation, the study looks at different tether materials and shapes. The goal is to identify those that are best for improving the survivability of tethered systems in Low Earth Orbit (LEO) and boosting the chances of successful tether deployment.

Next, the thesis looks into the choice of deployment and storage systems. It assesses their practicality and success in reaching the set goals. A comparison of various 3D simulation environments, such as Blender, Unity™, and NVIDIA IsaacSim, is carried out to find the best tool for creating the digital twin.

The thesis then introduces the concept of scaling factors, explaining how the dynamic behavior of a real tethered system in orbit can be replicated at laboratory scale using an air-bearing platform. This physical testbed can then be reproduced in a virtual environment.

Finally, the thesis describes how the experimental setup was created in Unity™, which includes modeling the air-bearing platform and the tether along with their interactions. Three different control strategies for the deployed tether case are implemented and analyzed: PID tension control, PID position control, and LQR position control. After validating the tether modeling methods, the simulation of the tether deployment phase up to a specified length is conducted. This is combined with a PID-based tension control algorithm. The thesis wraps up with a thorough analysis of the results and a discussion of the main outcomes.

1.2 Literature review on tethered satellite missions

Below is a brief summary of the missions that have involved tethered satellite systems to date. Much of the information cited in this section was taken from Mattia Li Vigni's bachelor's thesis [1].
Since 1966, various missions involving tethered satellite systems have been launched to study and test the feasibility of several ambitious objectives, including:

- **Propellant-less propulsion**: generating thrust (to increase orbit) or drag (to decrease orbit and deorbit) without using propellant. This is achieved through the interaction between a conductive tether and Earth's magnetic field.
- **Formation flying and tethered constellations**: connecting satellites constellations with tethers to maintain precise geometric configurations. Tethers can simplify formation control compared to propelled systems.
- **Electric power generation**: similar to electrodynamic tethers for propulsion, but the primary goal is to generate power for the connected satellite by exploiting current induced in the tether as it moves through Earth's magnetic field. This could serve as an alternative or supplement to solar panels, particularly for satellites operating in low orbits where magnetic interactions are more pronounced.
- **Space debris capture and removal with tethers**: A "chaser" satellite equipped with a tether approaches the debris. Once captured (various technological solutions, such as nets, are being studied for this purpose), the system (chaser, tether, and debris) can be

deorbited passively if the tether is non-conductive, by increasing drag, or actively if the tether is electrodynamic.

1.2.1 Gemini XI and Gemini XII

On September 12, 1966, NASA's Gemini XI made history as the first tethered space mission. The crew's main goal was to connect with the Gemini Agena Target Vehicle 11 (GATV-11) and see if they could create artificial gravity by spinning the two spacecraft around their shared center of mass. They successfully deployed a 30-meter tether and achieved a stable spin, producing a very small amount of artificial gravity, about 0.00015 g.

The next mission, Gemini XII, in 1967, aimed to study gravitational stabilization between the spacecraft and the Agena. Even though there was only a tiny gravitational difference between the two vehicles, the mission showed some level of gravity-gradient stabilization. It also highlighted the complex behavior of tethers in space, exceeding initial expectations.

1.2.2 TSS-1 and TSS-1R

The Tethered-Satellite System (TSS-1) marked the inaugural mission of NASA and the Italian Space Agency (ASI) collaborating on tethered satellite systems. Launched aboard the Space Shuttle Orbiter Atlantis (STS-46) on July 31, 1992, the mission's primary objective was to investigate the dynamics of the tether and analyze the electromagnetic interaction between the tether and the space plasma.
The mission's objectives included flying the Shuttle away from Earth, extending the 12 m deployment boom to provide a safe clearance between the probe and the Shuttle, and then deploying the 20 km tether. The deployment process began with thrusters, followed by gravity-gradient stabilization. The 2.54 mm diameter tether, constructed with a complex structure and various materials such as Nomex, copper, Teflon, and Kevlar, was designed to achieve electrical conduction and tensile strength. While the boom activation was successful, a bolt interference in the deployer mechanism prevented the complete deployment of the tether. The mechanism stopped while the tether was only 256 m long. Despite this setback, the mission still allowed for the exploration and analysis of certain deployment regimes.

In 1996, the Tethered Satellite System (TSS) had a second flight opportunity on Space Shuttle Columbia (STS-75). This time, the tether was successfully deployed and reached a length of 19.7 km, close to its full length of 20 km. However, just before the deployment was completed, the tether broke suddenly. An electrical discharge generated enough heat to melt it.

1.2.3 SEDS-1 and SEDS-2

The **first Small Expendable Deployer System (SEDS-1)** launched in 1993. It included an "end mass" payload satellite attached to the second stage of a Delta II rocket with a 20 km long tether. The main goal was to test situations where tether retrieval was not needed, using a simple deploy-only system. The tether deployed successfully, and after one orbit, the satellite and tether were put on a path to reenter the atmosphere. This allowed researchers to study how to use a tether to place a payload in a deorbit trajectory and observe its reentry.

In 1994, NASA launched **SEDS-2**, a mission similar to SEDS-1 but with an improved braking system. The mission's objectives included testing the efficiency of the close-loop deployment control law, which applied braking force based on the unrolling tether's speed to prevent bouncing. Additionally, it aimed to study the dynamic evolution of a tethered system over a long period and analyze the risk associated with micrometeoroid impacts. Despite the expected lifetime exceeding 20 days, the 19.7 km tether was severed a few days after launch, likely due to untracked debris or micrometeoroids. Even with this setback, the mission was considered successful. The end mass smoothly stopped along the local vertical, with minimal residual sawing, thanks to the implemented control law.

1.2.4 YES2

YES2 mission, launched in 2007 aboard a Foton capsule for the Foton-M3 mission, aimed to demonstrate the capability of returning a low-mass re-entry capsule (FOTINO) using a tether (known as "SpaceMail"). The YES2 tether hardware was essentially an evolution of the successful concept employed during SEDS missions. This system involves axial deployment of a tether from a static spool, which proved to be an extremely reliable concept. The **YES2 tether** was constructed from Dyneema (non-conductive), measuring 31.7 kilometers in length and 0.5 millimeters in diameter. **The 36-kilogram payload** consisted of three main components: FLOYD (Foton Located YES Deployer, weighing 22 kilograms), which housed the tether spool and was located within the Foton-M3 capsule; MASS (Mechanical and data Acquisition Support System, weighing 8 kilograms); and FOTINO, the re-entry capsule (6 kilograms).
After a 11-day launch, the YES2 mission was activated. MASS and FOTINO were deployed from FLOYD. While tether deployment faced some challenges, it was overall successful, with the tether reaching its full length.

1.3 Overview of deployment systems technologies and air-bearing platform experiments

1.3.1 Experiments on air-bearing inclinable turntable

In the context of the experiments carried out on air-bearing platforms, experiment [2] is of special interest for this work. It comprises a ground-based experimental setup engineered to simulate the deployment dynamics of tethered satellites in orbit. The system employs a tiltable air-bearing rotating platform, enabling individual adjustment of the tilt and rotational velocity. Some interesting aspects are:

- Scale factors: they have been calculated for the different forces which act on the secondary satellite. These factors are employed to compare the behavior of the tethered system in the experiment with that in orbit.
- Satellite attitude determination: to track the motion of the secondary satellite, two cameras are used. For this purpose, a program based on OpenCV has been developed.

The results suggest that the experimental setup is promising for the study of tethered satellite behavior and that the scaling factors have been implemented effectively. The experiment authors propose to implement control laws and model and measure the system damping.

1.3.2 Shape and material related differences

Up to present, the most commonly employed tether shapes have been cylindrical, tape and braided. Some studies, such as [3] and [4], have shown the superiority of the tape shape compared to the cylindrical one in different aspects.
Nevertheless, analyzing the study [5], we can state that, regarding shape, the braided shape is an interesting alternative, as is Spectra for materials. These options will be considered in the analyses conducted in the relevant sections of this thesis.

1.3.3 Deployment control laws

Distinct control laws have been implemented and tested so far:

- **Threshold-based control law**: the tether tension is measured, and based on its value, the release is either accelerated or slowed down.
- **Velocity control law**: it aims to maintain the deployment speed at a desired value.
- **Length-proportional tether deployment system**: the speed deployment is proportional to the length of tether that still needs to be deployed.
- **SEDS system control low**: this mission implemented a control law based on the velocity in order to obtain the desired tether tension during the whole deployment [6], [7].
- **YES2 system contol low**: this mission implemented different control laws during for each phase of the deployment. It used the "barberpole mechanism" for controlling the deployment of the tether, where the tether tension depends exponentially on the number of windings, while the friction depends on the diameter of the pole [6], [8].

In addition, most systems exploit the gravitational gradient in order to deploy and stabilize the tether. This technic requires an initial impulse (like a spring ejection) and an end mass.

1.3.4 Storage and deployment systems

Various storage and deployment systems are available, which will be further discussed in section 2.4 A concise overview is provided below.

- **Barberpole mechanism**: it's a simple and low-cost friction mechanism. The tether tension depends exponentially on the number of windings, while the friction depends on the diameter of the pole. [6]
- **Rotating spool**: the system rotates to wind or unwind the tether, it is controlled by managing the rotational speed of the spool, often through a motor with a braking system.
- **Stationary reel**: the reel onto which the cable is wound remains fixed. A separate element, often called a 'wire guide' or 'rotating arm', rotates around the fixed reel and allows the cable to unspool from one end.
- **Origami**: the tether is folded onto itself following a predefined pattern. The aim is to occupy the smallest possible volume. The deployment occurs by opening the folds of the cable: this can be activated by the release of constraints holding the folds closed, by the strain energy stored in the folded material, or by external forces.
- **Thrust motor systems**: they are a novel mechanism for deploying electrodynamic tethers. These systems utilize electric motors to unlock and eject a top plate, converting spring potential energy into kinetic energy for initial deployment. Designed to be low in volume and weight (e.g., 2.3 kg), these systems align with the evolving trends in space payloads. [9]
- **CubeSat DESCENT mechanism**: it's constituted by a conical spring and its supports positioned on the two CubeSats, a tether storage box (origami method) located on the deputy CubeSat, a tensioner and fastening cables. When the fastening cables are cut, the spring expands, generating an impulse force that separates the two CubeSats. This system does not include a braking system for the final part of the deployment. This method will not be covered in the analysis. [10]

1.4 3D simulation software for tethered satellite systems digital twin

The simulation of tethered satellite systems has very often required the development of specific software capable of reproducing the complex dynamics of these systems in space.

The motion of tethered systems is described by a set of nonlinear, non-autonomous, and coupled ordinary and partial differential equations. These equations create significant challenges for reliable prediction.

To tackle this complexity, researchers have developed several modeling approaches [11]:

- **Rigid Body/Dumbbell Model**: This is the simplest approach. It represents the tether as a rigid, often massless, rod connecting two point masses. It serves as a starting point and helps in understanding the basic effects of gravity gradient stabilization.
- **Lumped-Mass (Bead-and-Chain) Model**: In this approach, the tether is broken down into a series of interconnected point masses, springs, and dampers. This setup models the tether's flexibility, including both longitudinal and lateral vibrations. It also helps evaluate tether tension and how the spacecraft responds to tension peaks.
- **Continuous Thread Model**: This method treats the tether as a continuous flexible body. It derives its equations of motion from a force balance on a small part of the tether. The resulting partial differential equations are usually solved numerically, often using finite element methods.
- **Finite Element Method (FEM) / Absolute Nodal Coordinate Formulation (ANCF)**: These methods provide high-accuracy models of tether flexibility, deformation, slack, and rebound. They are particularly useful for complex situations like three-body systems or net capture dynamics.

Below, we analyze the most relevant software used in real missions. Most of the information about the software used was taken from [12].

SKYLINE: it is a tether simulation tool developed in the early 1990s, was designed to analyze the deployment phase of long tethers. It takes into account effects such as oscillations and aerodynamic drag. However, it does not include algorithms or control procedures for the deployed tether or the TSS.

YESSim: it is a tool created for the European tether mission Yes2. It is an improved version of BEASim, which modeled the tether as a lumped-mass system. This approach took into account various orbital disturbances. The tool mainly supported the design of the eject/retrial and brake system for the SEDS-1 and SEDS-2 missions.

While the "lumped-mass" approach simplifies the model, it is computationally efficient and usually good enough to capture the larger dynamics related to deployment and release.

STS simulator: developed in the 1990s, was designed to determine algorithms for retrieving or deploying the tether. A crucial feature of this simulator is its capability to apply external forces to the tether.

Matlab and Tether Dynamics Toolbox (TDT): many dynamic models for tethered systems, particularly those involving partial differential equations, are initially converted into a finite number of ordinary differential equations using the Ritz procedure. These resulting ordinary differential equations are then numerically integrated using **Matlab's ODE** solvers. Additionally, there's a specialized Matlab-based toolbox called the **Tether Dynamics Toolbox**

(TDT) that offers at least three distinct approaches for tether dynamics: the **rigid body model**, the **continuous thread model**, and the **lumped-mass approach**. The TDT features support for specifying tether parameters, analyzing dynamics with damping, evaluating required tether properties, pre-analyzing instabilities, and considering orbital perturbations for short missions.

1.4.1 Standard software for tethered dynamics simulations

Software	Key characteristics	Main goals
Tether dynamics toolbox [12]	Flexible modeling	Analysis of ADR missions
SKYLINE [12]	Deployment, Oscillations	Analysis of long tether deployment phase
YESSim [12]	Concentrated masses, Orbital perturbations	Design eject/retrial SEDS system
STS Simulator [12]	Deployment algorithms, external forces	Determination of tether control algorithms
MTBSim [13]	6 DOF, mission planification	Mission YES2 simulation, SpaceMail
MSC Adams [14]	Rigid/flexible bodies, joint modeling, external forces	Vehicle structure analysis, suspensions
Simscape Multibody (MATLAB) [15]	Flexible body dynamics	3D Mechanism modeling
EcosimPro [16]	Object-oriented modeling	Tethered satellite systems modeling
Basilisk [17]	Spacecraft-centered mission simulation framework	Modeling the attitude dynamics of multi-body spacecraft
NASA Trick Simulation Environment [18]	Creation of applications throughout all stages of spacecraft development	Simulations to support human spaceflight activities

Table 1.1: Standard software for tethered dynamics simulations

1.4.2 Unexplored 3D simulation software for tethered satellite systems

3D simulation software, such as **Blender**, **Unity™** or **NVIDIA IsaacSim**, could be extremely useful in the development of a digital twin for complex physical systems, with the aim of validating and testing algorithms and control strategies.

This type of software is characterized by **powerful physics engines** (in particular Unity™ and IsaacSim) which allow for an **accurate simulation of the real-world physics**, and they are able to compute step by step the system evolution, taking into account forces, masses, collisions, constraints and the behavior of joints with high fidelity. [19, 20] Furthermore, another relevant characteristic is the **opportunity of recreating different materials** and their properties, like static friction, dynamic friction, stiffness, etc. Hence, the physical fidelity offered by these software allows studying complex phenomena in a virtual, safe environment: for example, non-linear behaviors, accidental collisions, and instabilities could come out during simulation, just as it would occur in a real case, allowing engineers to identify them during the design phase.

The case of interest focuses on simulating an air bearing platform. This is a laboratory setup that creates an air cushion to mimic the nearly frictionless motion of a satellite on a plane. This kind of testbed provides an almost frictionless surface, supporting the floater on a thin layer of pressurized air and simulating microgravity in two dimensions. These planar platforms, often made from highly polished granite, allow small satellite simulators to move across the surface. This setup creates conditions similar to those of an orbiting satellite, which is useful for experiments in dynamics and control.

To reproduce a system like this in a 3D simulation, both the floater (its mass, dimensions, and other features) and the low-friction environment need to be modeled. Physics engines in 3D simulators can effectively replicate tether dynamics. They can approximate a tether as a series of rigid segments with movable joints or through flexible element models. This design allows the virtual catenary to respond to tension and gravity in a way that parallels a real tether.

Modeling tethers in this type of software has the advantage of offering a true-to-life visual and physical representation of tether behavior. It allows for the study of long-term dynamics or responses under control laws in the tether's deployed state. This understanding helps identify potential issues like tether vibrations, instabilities, or sudden slackening, which in turn helps develop necessary control measures.

Modern simulation environments provide a major benefit by allowing direct implementation of control laws on the virtual model. This allows for assessing their effectiveness. You can achieve this using scripting languages like C for Unity™ or Python/C++ for IsaacSim. Academic studies have shown the usefulness of this approach [20]. For example, a teleoperation platform was created using reinforcement learning algorithms on a virtual robotic arm in Unity™ [21]. The control successfully transferred to the real arm thanks to the accuracy of the simulation model.

Testing PID or LQR algorithms in the digital twin has many benefits. You can quickly adjust parameters, giving instant feedback on performance changes like oscillation damping, settling time after maneuvers, and tethered satellite stability. You can also introduce simulated disturbances to test the algorithm's resilience under extreme conditions.

In the context of satellite simulators on air bearings, an interactive 3D model lets you test maneuvers and control strategies before trying them on the physical platform. This approach ensures that the behavior observed in simulation mirrors the real-world situation. Tools like Blender, Unity™, and NVIDIA IsaacSim offer the perfect mix of physical realism and flexibility for digital twin applications in engineering. They allow for accurate recreation of complex experimental setups, such as tethered satellite systems, and enable pushing them to their limits in the virtual environment. This process provides valuable insights for design, control, and the success of future missions.

1.5 Research questions

In light of the problem presented, the objectives of this thesis are the following ones:

1. Select the material and the shape of the tether, paying particular attention to innovative solutions, such as new composites, in order to allow the tether to survive in the hostile low-atmosphere environment.
2. Analyze deployment systems and storage methods to identify the most suitable and reliable configuration. This configuration should ensure compatibility with the chosen tether and increase the likelihood of a successful deployment.
3. Compare the advantages and limitations of three different 3D simulation software (Blender, Unit™, NVIDIA IsaacSim) with the aim of selecting the most suitable one for achieving an accurate physical representation of an air-bearing platform.
4. Study the possibility of implementing different control laws for the deployed tether and the tether deployment in the selected software

Chapter 2

Optimal architecture for tethered satellite survivability and deployment reliability

2.1 Introduction to the AHP analysis

The Analytic Hierarchy Process (AHP) is an evaluation methodology for complex, multi-criteria decisions. In this context, it is used to weigh the most relevant material and design characteristics and compare competing alternatives. Each row of the final decision matrix corresponds to a specific criterion, with the first column reporting the weight assigned to that criterion. The remaining columns contain the normalized scores of the alternatives [22].

This chapter aims to select the architecture (material, shape, coating, and deployment system) that maximizes the survival of a tether satellite system in the LEO environment and that increases the likelihood of successful deployment. To achieve this, an Analytic Hierarchy Process (AHP) analysis is conducted for the material, shape, and deployment system, with coating excluded due to its single-configuration requirement. Qualitative properties are assessed using a 1-to-9 scoring scale, and the reference table 2.1 is used to complete the prioritization matrices. AHP is employed to analyze the candidate options for constructing the tether and perform a trade-off among their characteristics, ultimately identifying the configuration that best meets the mission needs. The workflow proceeds as follows:

1. **Initial prioritization**: an initial prioritization matrix is compiled by evaluating each criterion against every other using the provided scale and its reciprocals.

2. **Criteria weights**: The weights vector is the eigenvector linked to the maximum eigenvalue. It is normalized so that the sum of its components equals one.

3. **Alternative assessment**: Each alternative is evaluated against each criterion using measured data when available or a qualitative 1–9 grid when needed. The resulting score matrices for each criterion are column-normalized to sum to one.

4. **Overall scores**: Overall scores are calculated by taking the weighted sum across the criteria. This involves multiplying the score vectors for each criterion by the corresponding weights and summing the results. This produces the decision matrix and a ranking of the alternatives.

Intensity of importance	**Definition**	**Definition explanation**
1	Equal importance	Two activities contribute equally to the objective.
2	Weak or slight	
3	Moderate importance	Experience and judgement slightly favour one activity over another.
4	Moderate plus	
5	Strong importance	Experience and judgement strongly favour one activity over another.
6	Strong plus	
7	Very strong or demonstrated importance	An activity is favoured very strongly over another; its dominance demonstrated in practice.
8	Very, very strong	
9	Extreme importance	The evidence favouring one activity over another is of the highest possible order of affirmation.

Table 2.1: AHP fundamental scale: intensity of importance

2.2 Comparative study of tether materials

For what concern the material selection, the comparison focused on three very promising materials: Zylon, Kevlar, and Spectra. The alternatives are assessed against mission-relevant criteria to identify the most suitable tether material.

	Spectra (UHMWPE)	**Zylon HM (PBO)**	**Kevlar (Aramid)**
Flexibility (Young modulus) [GPa]	117 [23]	270 [24]	112.4 [25]
UV resistance	Qualitative score: 9 Excellent [26]	Qualitative score: 1 Poor [27]	Qualitative score: 1 Poor [25]
Chemical resistance	Qualitative score: 9 Very good [26]	Qualitative score: 5 Stable [27]	Qualitative score: 7 Good [25]
Atomic Oxygen resistance [cm^3/atom] [28]	3.74×10^{-24} High erosion	1.36×10^{-24} Moderate erosion	6.28×10^{-25} Low erosion
Max temperature for long time [°C]	70 [26]	650 [24]	77 [25]
Max temperature for peaks [°C]	130 [26]	650 [24]	482 [25]
Tensile strength [GPa]	2.59 [23]	5.8 [24]	3.6 [25]
Density [g/cm^3]	0.97 [23]	1.56 [24]	1.44 [25]

Table 2.2: Materials scores matrix

Regarding prioritization, the decision was made to give more importance to properties such as resistance to chemical agents, UV radiation, and atomic oxygen, compared to the mechanical properties of the tether, with the aim of extending as much as possible the survival of a tether.

	Flexibility	UV resistance	Chemistry resistance	AO resistance	T long	T peak	Tensile strength	Density
Flexibility	1	1/9	1/8	1/8	1/7	1/5	1/2	2
UV resistance	9	1	2	2	2	3	9	9
Chemistry resistance	8	1/2	1	1/3	1/6	1/5	7	9
AO resistance	8	1/2	3	1	1/5	1/3	7	9
T long	7	1/2	6	5	1	3	7	9
T peak	5	1/3	5	3	1/3	1	5	7
Tensile strength	2	1/9	1/7	1/7	1/7	1/5	1	4
Density	1/2	1/9	1/9	1/9	1/9	1/7	1/4	1

Table 2.3: Material prioritization matrix

The weights for the different criteria are the values of the eigenvector corresponding to the maximum eigenvalue:

0.021	0.258	0.094	0.127	0.284	0.170	0.029	0.015

Table 2.4: Material criteria weights

Finally, the decision matrix can be filled out:

Properties	Weights	Spectra	Zylon	Kevlar
Flexibility	0.021	0.009	0.004	0.009
UV	0.258	0.211	0.023	0.023
Chemistry	0.094	0.040	0.022	0.031
AO	0.127	0.013	0.036	0.078
T long	0.284	0.025	0.232	0.027
T peak	0.170	0.0175	0.087	0.065
TS	0.029	0.007	0.014	0.009
Density	0.015	0.007	0.004	0.005
TOTAL SCORE		0.329	0.423	0.247

Table 2.5: Material decision matrix

The analysis determined that **Zylon is the best compromise** with respect to the different properties and their relative weights.

2.3 Comparative study of tether shapes

Zylon was identified as the optimal material through the previous AHP analysis. Despite its exceptional mechanical and thermal properties that make it ideal for load-bearing applications in space, Zylon has significant environmental vulnerabilities. This implies that Zylon's environmental sensitivities are not minor details, but fundamental design challenges, necessitating that the "optimal" tether shape for Zylon facilitates or enhances the effectiveness of protective measures against UV and atomic oxygen.

Although Zylon's material properties are excellent, the shape of the tether is vital for achieving its full potential. The goal now is to assess how cylindrical, tape, and braided tether shapes affect the overall performance of the system in space. Each tether design has its own advantages and disadvantages related to key performance factors. These include structural strength, resistance to impacts from micrometeoroids and orbital debris (MMOD), vulnerability to UV damage and atomic oxygen, and the challenges of deployment and stability control. Braided and tape shapes involve twisting or folding, which can create stress points, especially at knots or folds. Cylindrical wires, while more straightforward, do not have redundancies. The larger shape (whether cylindrical, ribbon, or braided) affects how stress is distributed on a smaller scale.

2.3.1 Coating influence on UV degradation and environmental protection

Various coatings have been tested with a Zylon tether to date, including Photosil, nickel metallization, and TOR-LM polymer. Key studies on this topic include [29] and [30], which yielded the following results.

- **Nickel coating**: this coating can withstand atomic oxygen, but it has cracking problems due to thermal cycling. This happens because of the difference in thermal expansion with Zylon. It also has ferromagnetic properties that need consideration.
- **TOR-LM**: this coating did not provide complete protection, particularly against UV rays.
- **POSS**: These are organic-inorganic hybrid materials. When they come into contact with atomic oxygen, they form a thin and protective silicon dioxide (SiO_2) layer. In tests on Zylon fibers, sol-gel coatings, like 'Photosil', have reduced erosion, achieving a coated braid AO reactivity of about $\sim 1.7 \times 10^{-24}$. However, using chemical application methods can cause microcracks. A 10-20% POSS coating, just a few micrometers thick, on the fiber would offer significant protection against atomic oxygen and UV light while adding minimal weight.

Based on the studies mentioned, it can be concluded that **the best solution for protecting Zylon from UV rays and atomic oxygen (AO) is to use a POSS protective coating**. However, the application process, such as coating or plasma deposition, must be carefully controlled to avoid damaging the fiber.

2.3.2 Shape influence on space debris and micrometeoroids impacts

Cylindrical Tethers (Single Strand): this basic shape is particularly vulnerable to single-point failures. Even impacts from small debris can easily sever a thin cylindrical tether, leading to low survivability. Although hollow cylindrical tethers have been proposed to increase diameter and improve robustness against small debris, their fundamental single-strand nature remains a concern [31].

Tape tethers: their wider and thinner cross-section offer better resistance to MMOD (micrometeoroid and orbital debris) impacts compared to cylindrical wires. Instead of being severed, small debris impacts tend to create holes in the ribbon, allowing the tether to withstand critical impacts. The damaged area can be larger than the projectile's path due to the high temperature and stress upon impact. However, if a ribbon tether twists into loops, a single impact can potentially cause a cascading effect with multiple collisions [32].

Braided Tethers: this shape provides the highest level of MMOD resistance because of its structural redundancy. It consists of multiple interwoven strands, which create many load paths. If one strand is cut, the load shifts to the other strands. This process prevents total tether breakage.

2.3.3 Shape influence on deployment, stability and control

Tether deployment is a critical and complex issue: factors like the libration angle and tension control are crucial for stable deployment. Gravitational gradient forces naturally align tethers with the radial direction, but the system can oscillate (librate) like a pendulum: controlling these librations is essential for stability.

- **Cylindrical Tethers**: these are simpler to handle and less prone to twisting during deployment. However, single-strand tethers have experienced deployment failures (TSS-1) due to mechanical issues. [33]
- **Tape tethers**: These can be prone to twisting and forming non-ideal shapes during deployment and operation. This twisting can reduce the effective drag area. [34]
- **Braided Tethers**: Multi-strand braided structures are robust in terms of deployment, as demonstrated by the [35] tests.

2.3.4 Comparative performance overview and shape selection

In order to select the shape that best satisfies the objectives, the following AHP analysis was conducted.

	Cylindrical	**Tape**	**Braided**
MMOD resistance	Qualitative score: 1 Low	Qualitative score: 5 Medium	Qualitative score: 6 Medium
Susceptibility to UV degradation	Qualitative score: 1 High, entirely dependent on external coatings	Qualitative score: 1 High, due to the wide exposed surface	Qualitative score: 1 High, but braids offer self-shielding
Ease of deployment	Qualitative score: 9 Simple, but with risks of mechanical failures	Qualitative score: 1 Complex, due to the tendency for twisting and tangling	Qualitative score: 1 Complex, requires precise control
Deployment stability	Qualitative score: 9 Low tendency for twisting	Qualitative score: 1 High tendency for twisting	Qualitative score: 5 Complex dynamics
Manufacturing complexity	Qualitative score: 1 Low	Qualitative score: 5 Medium	Qualitative score: 9 High
Manufacturing cost	Qualitative score: 1 Low	Qualitative score: 5 Medium	Qualitative score: 9 High

Table 2.6: Shape scores matrix

	MMOD resistance	UV susceptibility	Ease of deployment	Deployment stability	Manufacturing complexity	Manufacturing cost
MMOD resistance	1	1/2	4	3	9	9
UV susceptibility	2	1	4	3	9	9
Ease of deployment	1/4	1/4	1	2	7	7
Deployment stability	1/3	1/3	1/2	1	5	5
Manufacturing complexity	1/9	1/9	1/7	1/5	1	3
Manufacturing cost	1/9	1/9	1/7	1/5	1/3	1

Table 2.7: Shape prioritization matrix

The weights for the different criteria are the values of the eigenvector corresponding to the maximum eigenvalue:

0.303	0.379	0.148	0.110	0.035	0.024

Table 2.8: Shape criteria weights

Finally, the decision matrix can be filled out:

Properties	Weights	Cylindrical	Tape	Braided
MMOD resistance	0.303	0.025	0.126	0.151
UV susceptibility	0.379	0.126	0.126	0.126
Ease of deployment	0.148	0.121	0.013	0.013
Deployment stability	0.110	0.066	0.007	0.037
Manufacturing complexity	0.035	0.027	0.005	0.003
Manufacturing cost	0.024	0.018	0.004	0.002
TOTAL SCORE		0.384	0.283	0.333

Table 2.9: Shape decision matrix

The analysis determined that **the cylindrical shape is the best compromise** with respect to the different properties and their relative weights.

2.4 Comparative study of deployment systems

Various mechanisms have been employed for tether deployment, each with distinct advantages and disadvantages that influence their suitability for specific mission profiles and tether characteristics. These can be broadly categorized into **friction-based deployment systems**, such as the Barberpole, **spool-based deployment systems** (either rotating or stationary), and systems that rely on stored energy from folding the tether for storage, like the **Origami method**. Another promising type of mechanism currently under study is **based on a thrust motor**.

Below is a brief overview of these different deployment systems, followed by an AHP analysis comparing the various proposed solutions to determine the system best suited for the defined objectives and for the tether being studied.

2.4.1 Barberpole system

The Barberpole is a friction braking mechanism where the tether is wound around a simple pole to control the deployment speed by regulating tension. Through a toothed wheel, it allows for control of the tether's entry angle into the mechanism, and thus the exit tension. Friction depends exponentially on the number of wraps, allowing for a wide range of tension levels (from 10mN to 3N for YES2), which is crucial for controlling dynamics over variable tether lengths (e.g., from 300m to 30km). An advantage of this system is that it consists of few moving parts and few tether guides, enhancing reliability and safety and reducing the risk of jams or tether damage. A disadvantage of this system is that tether coils can accumulate near the entrance or overlap, especially with a high number of wraps, narrow poles, low roughness, or low input tension, potentially reducing friction or causing tangles [6]. The Barberpole, while a tension control mechanism, could be integrated into a retrieval system (acting as a brake/regulator), but it is not a complete storage and retrieval system itself.

2.4.2 Rotating spool system

Spool methods are very common for tether storage and deployment. Rotating spool systems involve the entire spool rotating. This is a well-known technology, commonly used for various applications like ropes and tapes. A major advantage of this type of mechanism is that it works independently of the tether's cross-sectional shape, so it could be used without major issues even with a braided tether. However, one must consider the inherent complexity due to the need for spool supports and bearings, which can compromise overall conduction and thermal performance if not designed correctly [36]. Rotating spool systems are inherently bidirectional, allowing the spool, driven by a motor, to rotate in both directions. This enables controlled deployment and retrieval of the tether. The spool's design ensures neat winding of the tether, which is crucial for retrieval.

2.4.3 Stationary spool system

In this type of system, the spool is fixed, and a separate element, often called a "wire guide" or "rotating arm," rotates around the spool, allowing the tether to unwind. It has smaller dimensions compared to the rotating spool system and does not require spool supports or bearings, thus having a simpler mechanical design. A critical aspect that must not be overlooked, and which represents the biggest problem in using this type of system, is that it introduces significant torsional rotation (360° per turn), leading to thousands of turns for kilometers-long tethers. This can be problematic for tether integrity [36].

2.4.4 Thrust motor deployment system

As the name suggests, this is a new mechanism for deploying electrodynamic tethers based on thrust motors, designed to overcome historical high failure rates. It uses electric motors to unlock and eject a top plate, converting the spring's potential energy into kinetic energy for initial deployment. It is designed for low volume and weight (e.g., 2.3 kg), aligning with future trends in space payloads. Thrust motor system is a passive expulsion system, meaning the deployment speed or tether tension cannot be regulated and it cannot refold or restore the tether to its original configuration after deployment. [37]

2.4.5 Origami deployment system

This method involves folding the tether. It is generally not compact enough and unsuitable for tethers kilometers long that retain fold memory. This implies that for long tethers and materials with significant bending memory or stiffness, this approach is impractical [36]. With Origami system, there is no built-in way to put the tether back into its pre-deployment configuration.

2.4.6 Comparative overview of features and deployment system selection

The criteria to be taken into consideration in the AHP analysis are:

1. Active control
2. Compatibility with the chosen tether
3. Reliability
4. Roll-in capability

	Barberpole	**Rotating spool**	**Stationary spool**	**Thrust motor**	**Origami**
Active control	Qualitative score: 9 Yes	Qualitative score: 9 Yes	Qualitative score: 9 Yes	Qualitative score: 1 No	Qualitative score: 1 No
Tether compatibility	Qualitative score: 9 All tethers	Qualitative score: 9 All tethers	Qualitative score: 9 All tethers	Qualitative score: 4 Not all tethers	Qualitative score: 1 Only tape tethers
Reliability	Qualitative score: 5	Qualitative score: 9	Qualitative score: 7	Qualitative score: 3	Qualitative score: 3
Roll in	Qualitative score: 1 No	Qualitative score: 9 Yes	Qualitative score: 9 Yes	Qualitative score: 1 No	Qualitative score: 1 No

Table 2.10: Deployment systems scores matrix

	Active control	Tether compatibility	Reliability	Roll in
Active control	1	1/3	1/8	9
Tether compatibility	3	1	3	9
Reliability	8	1/3	1	6
Roll in	1/9	1/9	1/6	1

Table 2.11: Deployment systems prioritization matrix

The weights for the different criteria are the values of the eigenvector corresponding to the maximum eigenvalue:

0.131	0.468	0.367	0.033

Table 2.12: Deployment systems criteria weights

Finally, the decision matrix can be filled out:

Properties	Weights	Barberpole	Rotating spool	Stationary spool	Thrust motor	Origami
Active control	0.131	0.041	0.041	0.041	0.004	0.004
Tether compatibility	0.468	0.132	0.132	0.132	0.059	0.015
Reliability	0.367	0.068	0.122	0.095	0.041	0.041
Roll in	0.033	0.002	0.014	0.014	0.002	0.002
TOTAL SCORE		0.242	0.309	0.282	0.105	0.0615

Table 2.13: Deployment systems decision matrix

The analysis determined that **rotating spool system is the best compromise** with respect to the different properties and their relative weights.

2.5 Summary of the selected architecture

In conclusion, following the comparative analyses and technical evaluations performed, we can confirm that the system architecture best satisfying the mission requirements has been fully identified.

The chosen solution features a Zylon tether with a cylindrical shape, covered by a POSS (Polyhedral Oligomeric Silsesquioxane) surface coating. This combination was selected for its strong resistance to atomic oxygen and ultraviolet radiation, which are both crucial for the system's survival. For the deployment subsystem, a rotating spool mechanism has been chosen. This technology is the only one that meets all requirements and provides the essential ability to retrieve the tether if necessary. Thus, the selected design is the option that best meets the goals while adhering to the constraints.

2.6 Assessment of environmental and dynamic risks for tethered satellites in LEO

2.6.1 Risk identification

Space tethers are susceptible to several environmental and dynamic phenomena in orbit. Specifically, risks can be devided into:

- **Environmental risks**: space debris, micrometeoroids, ionizing radiation, thermal fluctuations, atomic oxygen, ionospheric plasma
- **Dynamic risks**: mechanical jams, uncontrolled oscillations, or collisions with the mother vehicle

Space debris and micrometeoroids impacts (MMOD)

Low Earth Orbit (LEO) is populated by myriad artificial debris and high-velocity meteoroid dust particles. Even a small object can sever a thin tether: studies estimate approximately one cut per km · year of exposure for a 1 mm wire in LEO [38]. Approximately half of the orbital tether experiments have failed due to micrometeoroid impacts [39].

- Potential impact: high – a single cut severs the tether, leading to mission loss
- Possible countermeasures: utilize redundant designs, robust materials, conduct durability tests, and avoid debris-dense orbits

Ionizing radiations-induced degradation

Space radiations that can most impact the life of tethers include cosmic rays, proton/electron beams trapped in the Van Allen belts, and solar flares. This radiation can degrade materials (especially insulators or polymers) and progressively weaken the tether. Furthermore, high electrical tensions (in conductive tethers) combined with radiation can cause unforeseen electrostatic discharges that damage the insulation. For example, the STS-75 (TSS-1R) mission highlighted how insufficient protection led to short circuits with the ionospheric plasma and the tether's breakage [40].

- Potential impact: medium – radiation does not instantly cut the tether, but it reduces mechanical/insulating strength over time. Electrical malfunctions (arcing) can be more critical
- Possible countermeasures: use insulating materials with high radiation resistance, apply protective coatings on exposed parts (e.g., dielectric films), insert local shielding if necessary, monitor current and adopt charge dissipators (plasma contactors)

Thermal fluctuations and atomic oxygen erosion

The tether experiences drastic temperature cycles, and these thermal excursions can cause expansion/contraction that may generate micro-fractures in composite materials. Furthermore, at LEO altitudes, residual atomic oxygen rapidly oxidizes many polymers. Extreme temperatures therefore affect mechanical strength and electrical conductivity, and repeated thermal cycling can weaken the tether, leading to internal cracks [40], [38], [41].

- Potential impact: medium – progressive degradation of tether properties, in extreme cases, the tether can lose elasticity or break due to thermal fatigue. The combined action of solar UV and atomic oxygen is degenerative for many plastic or polymeric materials.
- Possible countermeasures: use materials that can withstand high and low temperatures. Apply reflective or highly emissive coatings, such as silica or alumina, to reduce solar heating. Use anti-atomic oxygen paints as protective coats. Test thermo-mechanical behavior thoroughly under extreme conditions.

Ionospheric plasma interactions

A conductive tether inserted into the ionospheric plasma generates electrodynamic currents (Lorentz effect). The presence of electrical charges in the environment can lead to arcing phenomena or electrostatic discharges on the tether. [40]

- Potential impact: high, a discharge can sever the tether or render its electrical function unusable, potentially even short-circuiting the cable to the vehicle. Magnetic perturbations can induce voltage oscillations.
- Possible countermeasures: use suitable plasma contactors, like ion or electron emitters, to keep the tether neutral. Choose insulators that are thick enough and made from materials with strong dielectric properties. Remove static charges. Ventilate or fill high-voltage containers with inert gases to avoid internal plasma formation.

Mechanical jamming

The controlled release of the tether is essential. A sudden jam in the deployment mechanism, such as excessive friction or a reel lock-up, can send the end mass back, which risks a collision with the mother satellite. [42]

- Potential impact: high, an immediate jam blocks deployment and can damage both the cable and the satellite through direct collision. The mission fails if the tether is not extended correctly.
- Possible countermeasures: design deployment mechanisms with low friction and calibrated brakes, install real-time tension and position sensors, adopt automatic shutdown and rollback controls in case of unexpected spikes, deploy the tether gradually under software supervision to detect any jamming, and conduct long-duration ground tests, including on a digital twin.

Dynamic Instability

The free tether creates a "pendulum" system with the two masses at its ends. In-plane and out-of-plane oscillations, known as gravitational librations, can increase if they are not damped. Sudden changes in tension can lead to transverse and longitudinal vibrations. If these vibrations are not controlled, they can raise tether tension and lead to a loss of system attitude. [42]

- Potential impact: medium-high. Pronounced oscillations can cause the payload to hit the vehicle or stress the tether too much. Uncontrolled dynamics often lead to breakage or secondary collisions.
- Possible countermeasures: use dampers to reduce initial vibrations, implement active tension control to lessen movements, and perform dynamic simulations with lumped-mass models.

Severity	Score	Description	Performance
Negligible	1	Negligible impact	Minimal or no impact
Significant	2	Low impact, easily manageable	Limited effect, recoverable
Major	3	Significant impact	Performance degradation
Critical	4	Severe impact	Severe damage or partial loss
Catastrophic	5	Catastrophic impact	Loss of the system or mission

Table 2.14: Severity of consequence scoring scheme

Probability	Score	Description
Minimum	A	Qualitative: very unlike to occur Quantitative: $\leq 20\%$
Low	B	Qualitative: Not likely to occur Quantitative: 20-40%
Medium	C	Qualitative: May occur Quantitative: 40-60%
High	D	Qualitative: Highly likely to occur Quantitative: 60-80%
Maximum	E	Qualitative: Nearly certain to occur Quantitative: 80-100%

Table 2.15: Probability scoring scheme

ID	Category	Risk	Probability	Severity
EN-01	Environmental	Space debris and micrometeoroids impacts	C	5
EN-02	Environmental	Ionizing radiations-induced degradation	E	3
EN-03	Environmental	Thermal fluctuations	D	3
EN-04	Environmental	Atomic oxygen erosion	E	3
EN-05	Environmental	Ionospheric plasma interactions	C	4
DY-01	Dynamic	Mechanical jamming	D	4
DY-02	Dynamic	Dynamic instability	D	3

Table 2.16: Risks identification

E-Maximum			EN-02, EN-04		
D-High			EN-03, DY-02	DY-01	
C-Medium				EN-05	EN-01
B-Low					
A-Minimum					
Probability/ Severity	**1-Negligible**	**2-Significant**	**3-Major**	**4-Critical**	**5-Catastrophic**

Table 2.17: Risks matrix before mitigation

2.6.2 Risks mitigation

The AHP analysis on various materials, the comparison of three different forms, and the assessment of several possible deployment systems allowed us to determine the best combination of material, form and deployment system for the defined objectives: a **cylindrical Zylon tether** with a **POSS coating**, and a **rotating spool system** as deployment system. Below, we analyze how this choice helps mitigate the previously identified risks.

Space debris and micrometeoroids impacts

Zylon has exceptionally high tensile strength and excellent toughness. This makes it very resistant to debris and micrometeoroid impacts compared to less robust materials.

Ionizing radiation-induced degradation

Zylon (PBO) is an organic polymer, so it can undergo long-term degradation due to ionizing radiation, however using a protective POSS coating significantly lowers the severity of this risk.

Thermal fluctuations

Zylon has a very low coefficient of thermal expansion (CTE), so it undergoes minimal dimensional changes with temperature variations, which reduces thermal stress and tension fluctuations in the tether. The study [43] conducted at 295 K (room temperature) and 77 K (liquid nitrogen) have revealed that the Zylon fibers within the composite maintain an exceptionally high Ultimate Tensile Strength (UTS), surpassing 4.3 GPa at these temperatures. This drastically lowers the severity of the event.

Atomic oxygen erosion

Zylon is susceptible to atomic oxygen (AO) degradation in Low Earth Orbits (LEO). AO breaks chemical bonds on the polymer's surface. The POSS protective coating contribute to reduce the severity of this risk.

Ionospheric plasma interactions

Zylon itself is not conductive, so it does not directly interact with plasma regarding current generation or electrodynamic forces. Because of this characteristic, the interaction is limited to surface charge that can accumulate.

Mechanical jamming

Zylon's high strength and flexibility reduce the likelihood of breakage or damage during deployment. Mitigation here heavily depends on the deployment system design: controlled deployment is crucial to prevent "jamming". Mechanical jamming can be caused by several factors, including **tether creep**, **excessive friction** between the spool and the tether, and potential **uneven expansions or contractions** of the tether due to thermal fluctuations. These issues can lead to small undulations or bulges that hinder the tether's release. Zylon demonstrates practically zero creep or 0% unrecoverable deformation even under significant static loads, such as 40-58% of its ultimate breaking strength. This intrinsic property ensures that a Zylon tether, stored wound on a spool for extended periods, is highly unlikely to retain a significant "curvature" or coiled shape upon deployment. The elastic and viscoelastic components of deformation recover, leaving negligible permanent deformation [44]. To minimize excessive friction between the tether and the spool, a low-friction coating can be applied to the spool wall. Finally, to prevent excessive thermal fluctuations, ground-based thermal analyses must be conducted, and if necessary, thermal management solutions should be considered.

Dynamic instability

Zylon's lightweight nature helps maintain a low tether mass, which is generally favorable for dynamic stability. Its high elastic modulus and low creep (deformation under constant load) contribute to maintaining consistent mechanical properties. A cylindrical tether can have a simpler cross-section and a smaller exposed surface area compared to a braided one. This could lead to reduced drag effects and potentially instability, which must be accounted for in dynamic modeling.
Mitigating this risk requires complex dynamic analysis and the implementation of active or passive control algorithms.
Implementing a control law helps prevent overlaps or slippages that generate instability. The use of real-time tension sensors can help maintain uniform deployment. Another useful aspect for identifying critical instability points is to simulate the tether's dynamic behavior through digital twins and conduct small-scale tests in controlled environments, which is among the objectives of this thesis.

Using a cylindrical Zylon tether can therefore reduce the severity of risks related to MMOD, AO, and thermal fluctuations. Risks due to ionizing radiation and ionospheric plasma require a POSS protective layer to be mitigated. Risks during the deployment phase and those related to dynamic instability are mitigated by using a rotating spool system and implementing control laws for deployment.

For what concern the **probability** of events dependent on the space environment, it cannot be reduced, as it relies on natural phenomena.

The risk matrix below reflects the mitigation **attributed to the material, form, coating and deployment system selection**.

E-Maximum		EN-02			
D-High	EN-03	EN-04			
C-Medium		EN-01	EN-05		
B-Low			DY-02	DY-01	
A-Minimum					
Probability/ Severity	**1-Negligible**	**2-Significant**	**3-Major**	**4-Critical**	**5-Catastrophic**

Table 2.18: Risks matrix after mitigation

Chapter 3

Selection of 3D modeling software and methodologies for tether modeling

3.1 Digital twin software selection

For selecting the software environment in which to develop the digital twin of the air-bearing platform, three possible 3D simulation environments are analyzed: **Blender**, **Unity**, and **NVIDIA IsaacSim**. For each one, the relevant features and the pros and cons for use as the digital twin of the case study are discussed. The following requirements are considered:

- **Physical accuracy**: e.g., realistic modeling of the tether and its interactions with other objects in the scene
- **Scripting and integration of control algorithms**
- **Cross-platform portability**: ability to run the digital twin on different platforms
- **Maturity of the support community**

The option that best satisfies the defined objectives is first identified by evaluating the advantages and disadvantages of the three tools, and subsequently validated through an AHP analysis.

3.1.1 Blender

Blender is an open-source 3D graphics and animation software powered by the Bullet physics engine. It runs on Linux, macOS, and Windows, and it is completely free. Blender provides various built-in physics simulation tools (rigid bodies, soft bodies, constraints, etc.) integrated into its animation environment. In particular, the Bullet Physics engine enables rigid-body and constraint simulations, allowing, for example, chains of linked objects to represent a tether.

Advantages:

- **Free and open-source:** It is free to use and modify, and it has a large support community. Installation is easy, and the application does not require many resources.
- **Cross-platform support:** Blender works on Windows, MacOS, and Linux without any functional differences. This makes it easy to share the digital twin across different operating systems.

- **Python scripting**: it allows controlling nearly every aspect via Python. This characteristic offers an advantage for the specific use case, allowing custom logic (e.g., PID/LQR controllers and even a custom physics solver such as PBD or XPBD to compute tether tension) to be integrated with ease into the simulation. The Python API allows access to object properties (floater position, cable length, etc.) at each step and the application of custom forces or constraints.
- **Integrated physics simulation**: Bullet supports rigid bodies, collisions, and constraints such as hinges and joints. Tether elements can be modeled as rigid segments connected by spherical joints or fixed-distance links, yielding chain-like behavior to approximate a cable. flexible also supports soft-body/cloth simulations which, with appropriate settings, can represent flexible ropes. In general, a discrete cable model (e.g., a series of capsules/segments connected together) can be used to simulate tether physics.
- **Rendering and visualization**: if visual quality matters, Blender is great at producing realistic renders and animations. For a digital twin, it allows for detailed modeling of the platform and floater, as well as accurate visualization of system motions. This is useful for presentations or qualitative analysis.

Drawbacks:

- **Physical fidelity limited by Bullet**: While Bullet is reasonably accurate, it focuses on speed and stability in animations. It may need smaller time steps, more sub-steps, and more solver iterations to manage demanding situations like a very flexible or long tether. For instance, keeping stability in a long chain of segments can be difficult with an iterative solver like Bullet, especially if keeping simulation time reasonable is needed. In extreme cases, accuracy may not match that of newer engines, such as NVIDIA's PhysX 5, which is used in IsaacSim.
- **Handling complex interactions**: When deployed, the tether must first be wound around a pole. This needs to be handled through collisions between the cable segments and the pole or by using specific logic, which Bullet may only manage roughly. Achieving an accurate model of tether behavior in Blender is possible but requires extra effort.
- **Performance**: Running long simulations or many iterations in Blender via Python can be slower than in optimized environments. Blender is not primarily designed for high-speed interactive simulation loops; each frame computed with Python scripts adds overhead. Unity or IsaacSim may also leverage multicore CPUs or the GPU for physics more effectively.
  [45], [46]

3.1.2 Unity

Unity is a general-purpose game engine, widely used for interactive simulations and real-time 3D applications. It is proprietary software (free for personal/educational use below a certain revenue threshold), with broad cross-platform compatibility: it enables development and deployment on Windows, MacOS, and Linux. Unity integrates NVIDIA PhysX to handle rigid-body simulations, collisions, and joints.

Advantages:

- **Robust integrated physics engine**: Unity provides a strong out-of-the-box physics foundation via PhysX. Rigid bodies can be defined for the floater and other elements, forces can be applied, constraints (Unity supports hinge joints, fixed joints, spring joints, etc.) can be configured, and collisions detected. For the tether, a chain of small bodies connected by hinge joints with limits can yield flexible behavior. The PhysX solver is fast and stable in most cases, suitable for real-time game simulations and, with proper configuration, technical scenarios as well.

- **Interactive, visual environment**: the Unity Editor facilitates scene construction, import 3D models (e.g., floater and platform), place objects, and adjust physics parameters (mass, low friction for the air-bearing plane, tether joint lengths and stiffness, etc.) through a GUI. The simulation can be observed in real time in the Game view and modified quickly.
- **Flexible scripting (C)**: programming in Unity is mostly done in C. This language offers strong performance, often faster than Python scripts [47]. It's possible to implement PID and LQR control through scripts that check the floater position every frame and calculate control actions. Tether tension can be computed using methods such as XPBD implemented in C, which manually updates cable positions and constraints at each simulation step.
- **Performance**: the physics time step can be reduced to improve accuracy, though this increases CPU usage. Unity supports multithreading for some parts of physics and can efficiently handle a moderate number of bodies. Therefore, even without IsaacSim's GPU acceleration, Unity can run medium-complexity scenarios smoothly. In this case, with a single tether and a floater, Unity should perform well in real time.
- **Ecosystem and community**: Unity's large user base provides abundant resources: useful plugins/assets and extensive documentation. Unity also integrates with other platforms: there is ROS support if needed, and it's generally easy to communicate with external programs. This flexibility is valuable for exporting and analyzing data in other tools (e.g., MATLAB) or for future project expansions. Unity is already used in robotics for advanced visualization and simulations with human/VR interaction, supporting its suitability for digital-twin contexts [20].
- **Export as standalone software**: simulations can be run on another computer without Unity installed, by creating platform-specific builds. Unity's portability across target platforms is excellent.

Drawbacks:

- **Lower physical accuracy than specialized solutions**: although powerful, Unity's PhysX is designed for games. Some simplifications or limits may show up in demanding situations. For instance, simulating a long, flexible cable connected to a satellite requires high stability. PhysX (version 4.x in Unity) might introduce stretch or instability if the integration step is not small enough. NVIDIA made significant improvements in PhysX 5 (used in IsaacSim) for constraint stability, improvements that Unity has not integrated yet. This means that Unity may be slightly less accurate than IsaacSim for difficult simulations. In practice, this requires careful tuning of parameters, like multiple sub-steps, to achieve precise tether modeling without numerical oscillations.
- **Proprietary software and dependencies**: while free for many uses, Unity is not open-source. This implies less transparency into exactly how internal physics computations are performed.
  [48, 49, 20]

3.1.3 NVIDIA IsaacSim

NVIDIA IsaacSim is an advanced robotics simulation platform built on NVIDIA Omniverse. It is designed explicitly to create digital twins of robots and environments with very high visual and physical realism. It uses the latest generation of NVIDIA's PhysX engine (version 5) and leverages GPU hardware (RT Cores, CUDA) to accelerate simulations. IsaacSim targets AI and robotics scenarios: simulated sensors (RGB-D cameras, LiDAR), native ROS/ROS2 integration, synthetic data generation, etc. Although many of these features fall beyond the scope of the present work (e.g., sensors or machine learning), physical accuracy and tether modeling remain particularly relevant.

Advantages:

- **Maximum physical fidelity**: IsaacSim likely provides the most accurate physics of the three options. It uses PhysX 5, which brings better solvers for joints and constraints, reducing instabilities and unrealistic behavior. Overall, IsaacSim aims for believable and lifelike virtual environments, as stated in NVIDIA's documentation. This allows effects like tether dynamics to be solved with high precision, especially when using small time steps and GPU-accelerated solvers.

- **Python support and customization**: IsaacSim can be controlled through Python, much like Blender. It's possible to script the simulation, create custom controllers, and interact with the physics. For instance, we can access the world state, including positions, velocities, and forces, at each step and apply algorithms like XPBD-based tension calculation.

- **Cable/robotics-specific building blocks**: While there is not a dedicated "rope" module yet, IsaacSim offers examples and tools that help in constructing cables using multiple segments with joints. For instance, a rope demo mentioned in NVIDIA forums connects capsules with spherical joints to simulate a cable.

- **Scalability and GPU performance**: IsaacSim shines as simulation load increases. It can exploit the GPU for physics (PhysX GPU) to simulate thousands of objects or contacts in parallel. However, few bodies are required in this case.

Drawbacks:

- **High hardware requirements**: the main drawback is accessibility. To run IsaacSim optimally, a relatively recent and powerful NVIDIA GPU (ideally RTX series) is needed. Suggested minimum is roughly an RTX 3070 with 8 GB VRAM and at least 32 GB system RAM, plus tens of GB of disk space. This can be a serious barrier if we aim to run the digital twin on multiple machines without discrete GPUs. Notably, IsaacSim does not support non-NVIDIA GPUs, so it cannot run natively on modern Macs (Apple/AMD GPUs). Official support targets Windows 10/11 and Ubuntu 20.04/22.04, MacOS is excluded. Regarding accessibility (criterion: Windows, MacOS, Linux), IsaacSim fails to be truly cross-platform.

- **Installation complexity and software footprint**: IsaacSim is not a small program. It is usually installed through the NVIDIA Omniverse Launcher or Docker containers. The installation is large, with downloads over 10 GB. Starting the program can take several minutes because it loads many modules, such as RTX rendering, physics, and the user interface. In comparison, Blender opens in seconds, while Unity is faster than IsaacSim but still heavier. To use IsaacSim effectively, it is often necessary to adjust several settings, like PhysX GPU parameters. Users have mentioned this, especially when running multi-cable simulations. There is a lot of system requirement that might not be necessary for a simpler setup, such as a single tether and a floater on a plane.

- **Portability limitations**: sharing the digital twin requires the end user to have a compatible system and to install IsaacSim, reducing ease of distribution. There is no "standalone build" concept as in Unity; being a platform, anyone who wants to run the simulation must install the full environment. This conflicts with the requirement of being "easily launchable on different platforms".

- **Overkill without sensors**: IsaacSim shows its strengths when its unique capabilities are leveraged (physical sensors, RTX photorealism for vision training, complex multi-robot scenarios). In this context, only a subset of the available features would be relevant, as the simulator would primarily handle the mechanical dynamics of a cable and an object. Without the need for photorealism or sensing, the physical-fidelity advantage alone may not offset the software's greater complexity. Problems of this type can be effectively solved with lightweight tools, given appropriate parameter tuning, thereby avoiding the adoption of an enterprise platform.

- **Learning curve and maturity**: IsaacSim is still relatively new. Documentation is available but is still developing, and some bugs can appear, especially with GPU features and new capabilities. For instance, users have noticed instability when simulating multiple cables on the GPU and often have to switch to the CPU solver for stability. NVIDIA is rapidly improving the product, but using IsaacSim takes time to understand concepts like USD scenes and Omniverse extensions, as well as to identify hidden issues. Blender and Unity, being more established and popular, provide many more tutorials and forum discussions for common problems.

- **Less flexibility outside the standard path**: although Python is supported, IsaacSim is less open than Blender. For example, you don't have full control over the integrated PhysX core (you can configure it, but not easily replace it). Implementing your own XPBD solver inside IsaacSim may require disabling parts of the native tether physics and managing bodies manually via scripts, which is possible, but not exactly the workflow Omniverse is optimized for. Unity or Blender may offer greater freedom to "take control" of objects and move them arbitrarily according to external computations, whereas IsaacSim tends to steer you toward its internal system (which, if properly configured, should simulate the tether without writing XPBD from scratch).
  [50, 51, 52]

3.1.4 Assessment of the optimal solution

Summarizing the comparison, here are the key points:

- **Physical accuracy**: NVIDIA IsaacSim provides advanced physics capabilities (PhysX 5, GPU solvers) and is intended for applications of this nature; on paper it provides the best fidelity for tether simulation and free motion of the floater. Unity and Blender use more traditional engines (PhysX 4 and Bullet, respectively); both can reproduce system dynamics with good approximation, but Bullet may require stricter parameters (e.g., smaller time steps) to match the accuracy of Unity or IsaacSim.

- **Accessibility and Platforms**: Blender and Unity fully satisfy cross-platform requirements (Windows, macOS, Linux). Unity allows building applications for all these OSs from a single project, while Blender runs natively on all three. IsaacSim, by contrast, requires specific hardware (NVIDIA GPU), limiting accessibility. This factor alone may be decisive if the project must be shared with users on machines without discrete GPUs. In terms of ease of launch, Blender is the simplest (open the .blend file), Unity requires a platform-specific executable for users without the editor, and IsaacSim requires a full Omniverse installation. Thus, for accessibility, Blender and Unity clearly prevail.

- **Scripting and control integration**: Blender and IsaacSim both use Python for scripting. This language works well for implementing PID/LQR controllers and for XPBD-based tether calculations. Unity uses C, which is a high-performance language, along with a well-documented API. Unity's scripting is closely linked to the physics loop, or FixedUpdate. This makes it easier to synchronize controllers with the simulation.

- **Additional features**: In the perspective of a future expansion of the digital twin (for instance, through the addition of simulated sensors) Unity and IsaacSim provide greater possibilities, whereas Blender is less suitable for interactive extensions.
- **Community and support**: Blender and Unity have very large communities (Blender especially in graphics, Unity in gaming and increasingly in simulation/robotics). IsaacSim's community is smaller (mainly NVIDIA forums and technical docs). For troubleshooting or finding similar examples, Unity resources are likely the easiest to find (including robotics examples), like [21].

Considering all these aspects, **Unity offers the best balance** among accuracy, flexibility, and ease of use across different platforms. It provides good physical fidelity, especially with adjustable PhysX parameters and the option to add a custom XPBD solver for the tether. It is relatively easy to construct and visualize scenes, and it can work on any operating system without specific hardware needs. Additionally, it is built for interactivity and customization, which fits a digital twin managed by custom algorithms.

While its physical accuracy is not as high as IsaacSim's, Unity meets the required standards and presents a more practical option. It allows for the rapid development and testing of the tethered system while still achieving adequate realism. Blender, in contrast, would require more manual work to reach a similar level of simulation quality. IsaacSim, though physically superior, does not align well with the practical needs of the project, such as accessibility and ease of deployment.

Moreover, IsaacSim's extra features, like RTX photorealism, sensor simulation, and tight integration with the AI ecosystem, would not be fully used in this situation, all while adding significant complexity.

In conclusion, taking into account the necessary accuracy, portability across Windows, macOS, and Linux, the need for custom scripting, and user-friendliness, Unity stands out as the best choice for creating the digital twin of the air-bearing platform. It offers a balanced environment for implementing tether physics and PID/LQR control, with the benefits of a mature, cross-platform engine. Blender could be a suitable alternative if a completely free platform is desired and more time can be committed to manual simulation work. IsaacSim would mainly be considered if achieving maximum physical accuracy or transitioning to more complex scenarios becomes a priority; for now, based on the stated needs, it offers more than what is necessary.

3.1.5 AHP analysis

Furthermore, an Analytic Hierarchy Process (AHP) analysis was carried out to substantiate that Unity demonstrates the highest level of compliance with the defined objectives for the development of the digital twin, when compared with the other two candidate platforms.

With regard to the compilation of the prioritization matrix, the highest importance was assigned to physical accuracy, followed by the ability to implement control algorithms (which can be implemented in all three software platforms, but are assigned different qualitative scores depending on the efficiency of the programming language used).

	Blender	Unity™	NVIDIA IsaacSim
Physical accuracy	2	7	9
Accessibility	9	6	2
Scripting and control integration	3	9	3
Maturity of the support community	9	7	2

Table 3.1: Digital twin software scores matrix

	Physical accuracy	Accessibility	Scripting and control integration	Maturity of the support community
Physical accuracy	1	3	7	9
Accessibility	1/3	1	5	7
Scripting and control integration	1/7	1/5	1	8
Maturity of the support community	1/9	1/7	1/8	1

Table 3.2: Digital twin software prioritization matrix

The weights for the different criteria are the values of the eigenvector corresponding to the maximum eigenvalue:

0.564	0.286	0.116	0.034

Table 3.3: Digital twin software criteria weights

Properties	Weights	Blender	Unity™	NVIDIA IsaacSim
Physical accuracy	0.564	0.062	0.219	0.282
Accessibility	0.286	0.152	0.101	0.034
Scripting and control integration	0.116	0.023	0.070	0.023
Maturity of the support community	0.034	0.017	0.013	0.004
TOTAL SCORE		0.254	0.403	0.342

Table 3.4: Digital twin software decision matrix

As initially hypothesized, the analysis confirmed that **Unity** is the software solution most appropriately aligned with the project's objectives.

3.2 Implementation of scaling laws for experimental and digital twin representations

3.2.1 Scaling factors in the experimental setup

To ensure dynamic similarity between the real orbital environment, the laboratory test, and the scene in the digital twin, appropriate scaling factors must be introduced.

The granite bench facility acts as a simple physical model, in which the lengths, masses, and times are different from those of the reference orbital system. The Buckingham theorem guarantees that the correct ratios between tether stiffness, satellite mass, and characteristic accelerations are maintained in this model.

The Buckingham π theorem states that if a phenomenon is described by p variables involving q fundamental units, the entire system can be expressed using $r = p - q$ dimensionless parameters, known as π variables. By equating the π variables of the real case and those of the scaled case, the principle of similarity ensures dynamic similarity between the two systems. [53], [54], [55]

The system comprises a total of 15 variables:

$[x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z} \quad \ddot{x} \quad \ddot{y} \quad \ddot{z} \quad F_x \quad F_y \quad F_z \quad m \quad t \quad \omega]$

There are three fundamental units: length, mass, and time. Consequently, a total of 12 π variables can be identified and used to establish the similarity conditions between the two systems.

In particular, the scaling factors, namely λ_L, λ_m, and λ_t, are defined as the ratio between the real case and the scaled variables.

$$\begin{aligned} \lambda_t &= \frac{\tau}{t} \\ \lambda_m &= \frac{M}{m} \\ \lambda_L = \frac{\xi}{x} = \frac{\eta}{y} &= \frac{\zeta}{z} \end{aligned} \tag{3.1}$$

For the test with the tether fully deployed, the length is scaled by selecting $\lambda_L = 50$ (half of the actual 100 m tether mapped onto a 1 m cable). It is essential that the maximum real acceleration (approximately 0.1416 m/s^2) does not exceed the testbed's capabilities. Consequently, $\lambda_t \leq 20.3$, therefore $\lambda_t = 20$ is adopted.

3.2.2 Scaling factors in the digital twin

A highly convenient option is to design the digital twin scene at a 1:1 scale with respect to the experimental setup that needs to be replicated (in this case, the air-bearing platform discussed in the paper [53]). This approach eliminates the need for additional scaling factors between the experimental setup and the virtual scene. Instead, the digital twin simply incorporates the same scaling factors that were applied to ensure dynamic similarity between the real tethered satellite system and the air-bearing platform.

Three types of control will be implemented in the digital twin: PID tension control, PID position control, and LQR position control. The position controllers compute the "real" control acceleration based on the position error (in the coordinates of the tethered satellite system) and then convert it into a "scaled-table" acceleration. This scaled-table acceleration is then translated into the corresponding tilt angle of the table. Similarly, in the PID tension control, a force is computed, first converted into an equivalent acceleration in real coordinates, and then into the corresponding “scaled-table” acceleration. In addition, the desired position (and Ω for LQR control) can be provided relative to either the table frame or the real frame, selectable via a flag.

The scaling factor used to scale the acceleration is given by the following expression:

$$a_{bench} = a_{real} \cdot \frac{\lambda_T^2}{\lambda_L} \tag{3.2}$$

By applying this scaling factor, the acceleration on the table reproduces, with dynamic similarity, that of the real system.

It is important to note, however, that the scaling laws do not enforce a strict similarity of the tether stiffness. In fact, as also reported in [53], the physical tether employed in the laboratory was modeled using a cotton thread. This choice is motivated by the fact that cotton provides an effective approximation of the scaled stiffness within the measurable range of the load cell, while avoiding unrealistically high tension values that would arise if a high-performance material (like zylon, which was selected in the AHP analysis) were used at reduced length. For consistency with the experimental setup, the digital twin models the tether with the same cotton-like mechanical properties.

3.3 Methodologies for tether modeling

3.3.1 Problem formulation and modeling objectives

One of the most challenging aspects in the development of the digital twin lies in the modeling of the tether. Since Unity does not provide a preconfigured object for its representation, the tether must be implemented entirely through C scripting. The first step consists in discretizing the tether into a set of points connected by segments. Three main issues must then be addressed:

1. Updating the positions of the points, which evolve under the action of external and internal forces (modeled through Hooke’s law) acting on the tether
2. Graphically reproducing, in a manner consistent with its stiffness, the tendency of the segments to return to their equilibrium length as a consequence of the elastic force
3. Handling the collisions of the segments and points that constitute the tether with all the other elements of the scene

For each of the three problems, different solutions will be analyzed; specifically, Verlet integration will be compared with Euler integration, the PBD method with the XPBD method, and, finally, collision handling based on virtual spheres assigned to each tether node will be compared with virtual spheres placed at the center of each segment.

3.3.2 Euler and Verlet integration methods

Tether dynamics are highly sensitive to integration schemes and timestep sizes. Explicit integration methods, such as the Euler integration, can lead to instability in simulations. Euler method tends to accumulate error and energy within the system, resulting in non-physical behaviors which could lead to divergences or growing oscillations in the tether dynamics. Additionally, an ideally inextensible tether represents a highly stiff system.

The 3.5 table provides a comparison of the two methods with respect to several aspects relevant to tether modeling. As evident from the comparison, Verlet integration appears to constitute the most appropriate choice for the objectives of this study.

Feature	**Verlet**	**Euler**	**References**
Numerical stability	High stability for rigid constraints, allows larger timesteps	Unstable, requires very small timesteps	[56, 57]
Physical fidelity	Preserves segment length well	Tends to artificially stretch the rope	[58, 59]
Collision handling	Easier with XPBD	Harder, requires post-integration corrections	[60, 61];
Energy conservation	Good, avoid spurious energy drift	Poor, accumulates numerical error	[58, 56]
Cost per step	Update cost + constraint iterations (Gauss-Seidel/SOR)	Only update cost	[59]
Computational effort at equivalent visual quality	Large time step (dt), faster convergence	Small time step (dt), more substeps and iterations required	[59]

Table 3.5: Comparison between Verlet and Euler integration for tether modeling

Verlet integration is a numerical technique for the simulation of dynamical systems, particularly in the fields of computational mechanics and computer graphics. This method is based on a Taylor series expansion of the particle position, and it relies on an explicit integration scheme that updates particle positions using their values at previous time steps, thereby avoiding the direct computation of velocities. In practice, the position at the next time step is estimated from the current and the previous positions, resulting in reduced computational cost and improved numerical stability compared to other explicit methods such as Euler integration. It is characterized by several key features: it is simple to implement, stable over time, and naturally preserves physical quantities such as energy and angular momentum more effectively than other common integration schemes.

The starting point is the Taylor expansion of $x(t + \Delta t)$ and $x(t - \Delta t)$:

$$\begin{aligned} x(t + \Delta t) &= x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 + \mathcal{O}(\Delta t^3) \\ x(t - \Delta t) &= x(t) - v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 - \mathcal{O}(\Delta t^3) \end{aligned} \tag{3.3}$$

Adding the two preceding equations yields:

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + a(t)\Delta t^2 \tag{3.4}$$

The velocity is defined as:

$$v(t) \approx x(t) - x(t - \Delta t) \tag{3.5}$$

Substituting equation (3.5) in (3.4) yields:

$$x(t + \Delta t) = x(t) + v(t) + a(t)\Delta t^2 \tag{3.6}$$

As shown in the appendix section B dedicated to the code, this equation will be used to update the positions of the tether's points, which evolve under the action of external and internal forces.

3.3.3 PBD and XPBD methods

Position-based dynamic (PBD) is a method for real-time simulation of deformable bodies, such as tethers, in games and interactive applications. Its simplicity and robustness make it particularly attractive. However, the constraint stiffness is inherently dependent on the chosen time step and the number of solver iterations, which is the main limitation. Consequently, achieving nearly inextensible strings requires increasing the number of iterations, affecting not only the overall computational cost but also lacking a clear physical correspondence with material parameters. Furthermore, the effects of iteration count are non-linear, making it challenging to intuitively adjust parameters by simply rescaling stiffness values as a simple function or iteration count. For these reasons, PBD is confined to applications prioritizing computational speed over physical accuracy.

The **extended position-based dynamic (XPBD)** method extends PBD by introducing compliance α, the inverse of stiffness, and a 'total' Lagrange multiplier updated at each step. This allows the effective stiffness to become nearly independent of both the time step and the number of solver iterations, enabling direct specification of material properties (e.g., Young's modulus E) while preserving consistent behavior across different time steps and iteration counts. Additionally, XPBD provides a physically consistent estimate of the constraint force. [61]

The XPBD algorithm can be derived from Newton's equations of motion in the case where the force can be expressed as the gradient of a potential energy:

$$M\ddot{x} = -\nabla U^T(x) \tag{3.7}$$

within which an implicit-position level time discretization can be performed:

$$M(\frac{x^{n+1} - 2x^n + x^{n-1}}{\Delta t^2}) = -\nabla U^T(x^{n+1}) \tag{3.8}$$

the energy potential $U(x)$ may be further specified in terms of a vector constraint functions C:

$$U(x) = \frac{1}{2}C(x)^T\alpha^{-1}C(x) \tag{3.9}$$

where α is defined as the inverse of the stiffness:

$$\alpha = \frac{1}{k} \tag{3.10}$$

and we can define:

$$\tilde{\alpha} = \frac{\alpha}{\Delta t^2} \tag{3.11}$$

In this way, the dependence on the time step is 'absorbed' into the effective compliance. Additionally, the Lagrange multiplier is defined as:

$$\lambda_{elastic} = -\tilde{\alpha}^{-1}C(x) \tag{3.12}$$

Having introduced compliance, the elastic force can be reformulated in terms of the gradient of the potential energy as follows:

$$f_{elastic} = -\nabla_x U^T = -\nabla C^T \alpha^{-1} C \tag{3.13}$$

Through a series of substitutions into equations (3.8) and (3.13), the discrete constrained equations of motion are obtained, representing a nonlinear system:

$$M(x^{n+1} - \tilde{x}) - \nabla C(x^{n+1})^T \lambda^{n+1} = 0 \tag{3.14}$$

$$C(x^{n+1}) + \tilde{\alpha}\lambda^{n+1} = 0 \tag{3.15}$$

The objective is to identify an x and λ that fulfill the system's requirements. To solve this problem, we employ a linearization centered around the current state:

$$C(x_{i+1}) \approx C(x_i) + \nabla C(x_i)\Delta x \tag{3.16}$$

where $\Delta x = x_{i+1} - x_i$ and $\Delta\lambda = \lambda_{i+1} - \lambda_i$.
The linearization is substituted into equation (3.15):

$$C(x_i) + \nabla C(x)\Delta x + \alpha(\lambda_i + \Delta\lambda) = 0 \tag{3.17}$$

From equation (3.14), Δx can be expressed as: $\Delta x = M^{-1}J^T\Delta\lambda$. This expression can be substituted into the preceding equation. By factoring out λ and rearranging the terms, the following equation is obtained:

$$\Delta\lambda = -\frac{-C(x_i) - \alpha\lambda_i}{JM^{-1}J^T + \alpha} = -\frac{-C(x_i) - \alpha\lambda_i}{\sum_k w_k \left\|\nabla_{x_k} C\right\|^2 + \alpha} \tag{3.18}$$

where $d = JM^{-1}J^T = \sum_k w_k$ is a scalar referred to as the 'effective mass'. And the variable update takes the form:

$$\begin{aligned} \Delta x &= M^{-1}J^T\Delta\lambda \\ \lambda_{i+1} &= \lambda_i + \Delta\lambda \end{aligned} \tag{3.19}$$

The **case of a tether segment AB** is now considered, as it is instrumental for the implementation of the XPBD method in the code.

$$C(x) = \|x_B - x_A\| - L \tag{3.20}$$

$$\nabla_{x_A} C = -\hat{n}, \qquad \nabla_{x_B} C = \hat{n}, \qquad \text{with } \hat{n} = \frac{x_B - x_A}{\|x_B - x_A\|} \tag{3.21}$$

The effective mass becomes:

$$d = \sum_{k\in\{A,B\}} w_k \left\|\nabla_{x_k} C\right\|^2 = w_A \|\hat{n}\|^2 + w_B \|\hat{n}\|^2 = w_A + w_B \tag{3.22}$$

Thus:

$$\Delta\lambda = \frac{-\,C(x_i) - \alpha\lambda_i}{d + \alpha} \tag{3.23}$$

$$\Delta x = M^{-1}J^\top\Delta\lambda \tag{3.24}$$

$$x_A \leftarrow x_A - w_A\,\hat{n}\,\Delta\lambda \qquad x_B \leftarrow x_B + w_B\,\hat{n}\,\Delta\lambda$$

$$\lambda_{i+1} = \lambda_i + \Delta\lambda$$

3.3.4 Collision handling

The collision handling strategy should ensure that the tether interacts credibly with itself and the surrounding environment, avoiding interpenetrations and unrealistic behaviors. Two common approaches are to assign small virtual spheres to each node of the rope or to place spheres at the center of each segment connecting two nodes.

The main benefit of the first approach, spheres at each node, is its accuracy. Each point of the rope has its own "protective sphere." This setup ensures that collisions are captured clearly when the tether bends around complex obstacles or gets tangled. However, the computational cost goes up with the number of nodes. This leads to a quick rise in the number of pairwise checks. As a result, there can be jitter or instability because of many corrections happening at once.

The second approach, spheres placed at the center of each segment, aims to reduce the number of collisions to be handled. Each segment is approximated by a sphere (with a radius equal to half the segment's length), resulting in a slower increase in the number of colliders with rope resolution. This simplification leads to a more stable solver and improved performance, particularly critical in applications requiring smooth execution, however, it reduces the accuracy level too. In sharp bends or interactions with thin obstacles, the central sphere may not always accurately represent contact, causing the rope to penetrate objects or itself without detection by the physics engine.

In both approaches, when a penetration is detected, its depth is calculated. The endpoints of the segment are then displaced. This corrective procedure makes sure that the tether moves aside when it contacts external objects, preventing unrealistic passage through them.

Ultimately, the selection of the appropriate method depends on the specific intended use case. If visual fidelity and realistic responses to small environmental details are essential, per-node spheres are the better option, even though they require more processing power. On the other hand, if fluidity and simulation stability are prioritized, per-segment spheres offer a more efficient compromise.

In the specific scenario addressed in this work, since the tether is not anticipated to adopt highly intricate geometries necessitating high-precision collision handling, the design choice was to prioritize the stability and fluidity of the simulation. Consequently, collisions have been implemented using virtual spheres positioned at the center of each segment. Moreover, to prevent numerical artifacts due to an undersized diameter, a collision radius greater than the geometric one was used.

In summary, after evaluating various approaches, the following methods were selected to tackle the three primary challenges associated with modeling a tether in Unity: (i) Verlet integration, used to update the positions of the discretized tether points affected by external forces; (ii) the Extended Position-Based Dynamics (XPBD) method, used to ensure both a graphically and physically consistent representation of the tether, while accurately enforcing the segments' tendency to return to their rest length and enabling reliable tension computation; and (iii) a collision-handling strategy that uses virtual spheres, each with a radius equal to half the segment length and positioned at its center, to improve the overall fluidity and stability of the simulation.

3.4 Deployed tether and deployment phase: characteristics, challenges, and key aspects

The digital twin is designed to comprise two principal developments: the modeling of the deployment phase and the representation of the tether once it has been fully deployed.

The problem was tackled in two stages, each one more complex than the last. First, the digital twin was set up for the fully deployed tether. This was a simpler setup because of the known and constant free length. It allowed for early testing of modeling techniques and control algorithms without the added challenges of deployment dynamics. In this phase, the goal was to understand the basic behavior of the tether, including how tension is distributed, how it oscillates, and its stability.

After finishing this first phase, the more complex deployment process was introduced. This phase is different from the static scenario as it includes complicated dynamic and geometric aspects. The tether starts wound in a helical shape around the spool, adding spatial limits that need to be accurately reflected in the model. Another challenge comes from the gradual release of the tether segments. Each segment goes from being tightly wound to free flight upon release. This change affects the mass distribution and the system's overall response, requiring a more detailed representation in the digital twin. To simplify things, it was assumed that the segments still wound on the spool follow only kinematic rules, while once released, they are treated as dynamic elements subject to the full equations of motion.

In this phase, it is particularly important to observe dynamic transitions, like accelerations, whiplash effects, and sudden changes in tether tension that can occur during deployment.

Chapter 4

Scene creation in Unity™

4.1 Fundamental components of Unity™ for physical simulations

The construction of the digital twin within Unity was facilitated by a comprehensive set of fundamental objects and components, enabling both realistic physics simulation and graphical representation. Each component of the experimental setup, including the air-bearing platform, the floater, the granite bench, and the tether, was meticulously modeled by combining these Unity primitives with pertinent physical and visual attributes.

4.1.1 GameObjects

A GameObject is a generic container that represents any entity within a scene, and its functionality is defined by the components that are attached to it. In this project, every physical element, such as the floater, the granite bench, and each tether node, was implemented as a GameObject. This modular structure allowed for the independent management of geometry, physics, and behavior, while ensuring their coherent integration within the simulation.

4.1.2 Meshes and Mesh Colliders

To give GameObjects a real shape in the 3D environment, meshes were used. A mesh is made up of vertices, edges, and faces that define the shape of an object. To help detect collisions between the tether and these solid elements, mesh colliders were assigned to both the floater and the bench. While primitive colliders, like spheres, boxes, or capsules, use simple shapes to approximate geometry, mesh colliders copy the exact outlines of the object. This decision was crucial for ensuring realistic physical interactions, especially since the tether required high geometric accuracy when wrapping around the spool.

4.1.3 Rigidobies

The Rigidbody component serves as the central element in Unity's physics simulation, enabling GameObjects to be subjected to forces such as gravity, impulses, and collisions. In the digital twin, the floater was equipped with a rigidbody and assigned a mass of 1 kg, commensurate with the real experimental setup. This ensured that the floater responded realistically to the forces transmitted by the tether, as well as to interactions with the granite bench.

4.2 Simulation scene composition: implemented objects

Each scene element is meticulously modeled by combining the preceding primitives and properties.

4.2.1 Floater

The floater is the satellite mock-up moving on the air-bearing platform. In Unity, it was modeled as a cylindrical mesh with a diameter of 65 mm, offering a geometrical match with the physical prototype. To allow realistic physical interactions with the environment, a mesh collider was linked to the cylinder. A RigidBody component was added to the floater, with its mass set at 1 kg according to the experimental setup. This setup enabled Unity's physics engine to simulate how the floater responds to external and internal forces, especially the tension passed through the tether.

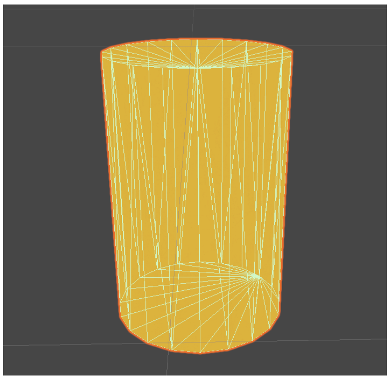


Figure 4.1: Floater in Unity

4.2.2 Granite bench

The granite bench was designed to mimic the flat microgravity environment of the physical testbed. It was modeled as a cube with dimensions of 630 mm x 400 mm x 80 mm. A mesh-type collider was assigned to this object, making it easy to detect contact between the tether and the surface. The bench itself stayed still, without a rigidbody, serving as support for the floater and tether to interact. This setup allowed the floater to move freely on the low-friction surface while following geometric constraints.

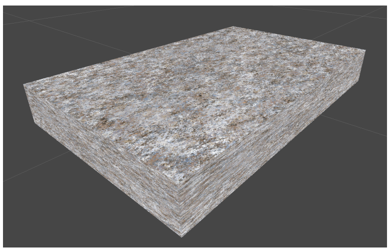


Figure 4.2: Granite bench in Unity

For both the floater and the bench, it was imperative to establish a material that would simulate the near-total absence of friction between the two bodies. In accordance with [62], the static and dynamic friction coefficients governing the interaction between the air-bearing and the granite table were assumed to be identical, with a value of $1 \cdot 10^{-5}$. Consequently, a virtual material with these characteristics was created, designated as "NoFriction," and assigned to both objects.

4.2.3 Tether

As discussed in detail in the dedicated chapter, the tether posed the most significant modeling challenge due to its flexible yet inextensible nature. In the virtual environment, the tether was discretized into a series of nodes connected by linear segments, effectively representing it as a mass–spring system. The two terminal nodes were anchored: one to the floater and the other to a pole (for the already deployed case), or to the rotating spool system (for the deployment case).

- **Mass distribution**: each node received an effective mass value. This value was calculated using the total mass of the tether and the selected discretization scheme.

- **Dynamics integration**: the Verlet integration method was adopted to update the positions of the intermediate nodes, chosen for its numerical stability and ability to conserve the inextensible property of the tether.

- **Constraints enforcement**: The tether segments were limited to keep their rest length, which is determined by the product of the tether's Young's modulus and cross-sectional area. The Extended Position-Based Dynamics (XPBD) algorithm was used to enforce these constraints. This approach addressed the instability problems found in standard position-based dynamics (PBD) when modeling stiff systems.

- **Collision handling**: to ensure correct interaction with the floater and the granite bench, each segment of the tether was sampled at multiple subpoints. These points were checked for intersection with colliders using spherical approximations. If a collision was detected, the nearest nodes were displaced outside the obstacle along the normal direction, thereby avoiding penetration while maintaining the geometric consistency of the tether model.

Furthermore, as previously discussed in the chapter on scaling factors, the scaling laws don't require a strict correspondence between the tether stiffness. The laboratory's physical tether was modeled with a **cotton thread** because it accurately approximates the scaled stiffness within the load cell's measurable range, avoiding excessively high tension if a high-performance material (like Zylon) were used at a reduced length. To maintain consistency with the experimental setup, the digital twin models the tether with the same cotton-like mechanical properties:

$$\text{Young Modulus E} = 4.8\,GPa$$
$$\text{Rope diameter} = 1.6\,\cdot 10^{-4}\,m$$
$$\text{Density } \rho = 1540\frac{kg}{m^3}$$

In [53], the diameter of the cotton thread is not explicitly specified; hence, it was assumed to correspond to the value reported above.
Finally, the tether was modeled with a length of 1.16 m in the pre-deployed case and 3 m in the deployment case. An additional 0.16 m, compared to the length reported in [53], was introduced to enable the floater to reach the edges of the granite bench.

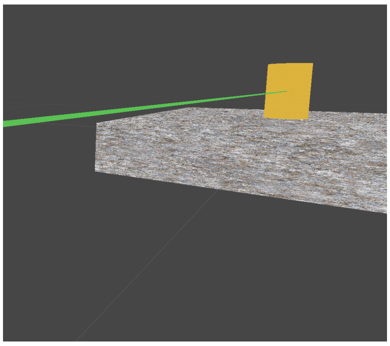


Figure 4.3: Tether in Unity

4.2.4 Pole and rotating spool sysem

In the case of the already deployed tether, it was sufficient to anchor the end opposite to the floater to a simple vertical pole, modeled in Unity as a cylinder. In contrast, for the deployment case, the rotating spool system (identified through an AHP analysis as the most suitable deployment mechanism for the defined objectives) was modeled as a horizontal cylinder, around which the tether is helically wound, and as a rigidbody, an essential attribute for handling both its rotation and its collisions with the tether.

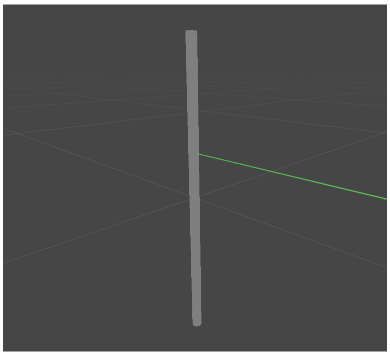


Figure 4.4: Pole in Unity

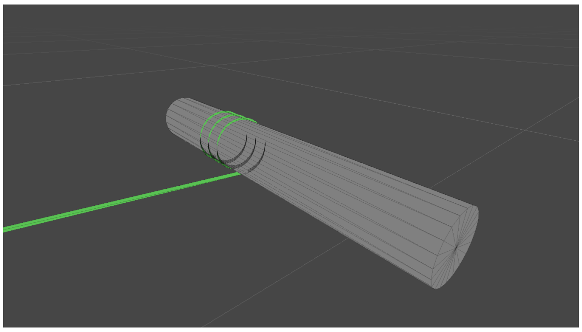


Figure 4.5: Rotating spool system in Unity

4.2.5 Full scene

Once the individual components were created and parameterized, they were integrated into a complete simulation environment. The floater, bench, and tether were positioned according to the physical configuration of the air-bearing testbed, for instance, the floater was placed at a distance of one meter from the pole.

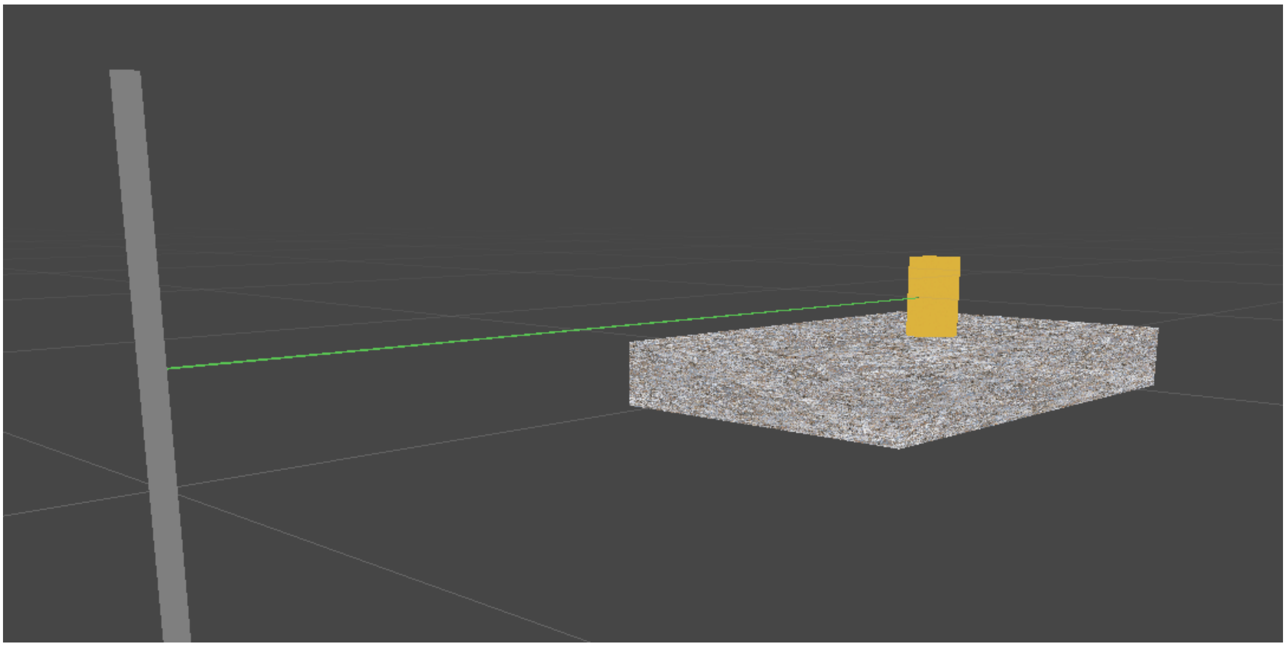


Figure 4.6: Full scene in Unity

Chapter 5

Software implementation of the deployed tether case

This chapter illustrates the implementation of the tether creation and management through software, as well as the implementation of various control algorithms, specifically: two position control algorithms (PID position and LQR position) and one tension control algorithm (PID tension). For a more detailed mathematical treatment of the Verlet and XPBD methods, reference is made to the preceding chapters.

5.1 Tether implementation - VerletTether.cs

The script utilizes a Verlet/XPBD (Extended Position-Based Dynamics) simulation method to create a physically realistic rope/tether simulator for Unity. The rope is represented as a series of point masses connected by inextensibility constraints, and its geometry is rendered using a LineRenderer. The solver employs Gauss–Seidel iterations and successive over-relaxation (SOR) techniques to enforce per-segment distance constraints. In addition, the script is responsible for handling the following aspects:

- Handling collisions
- Computing tension through the XPBD method
- Applying back-reaction forces to rigid bodies at the anchors
- Tether rendering

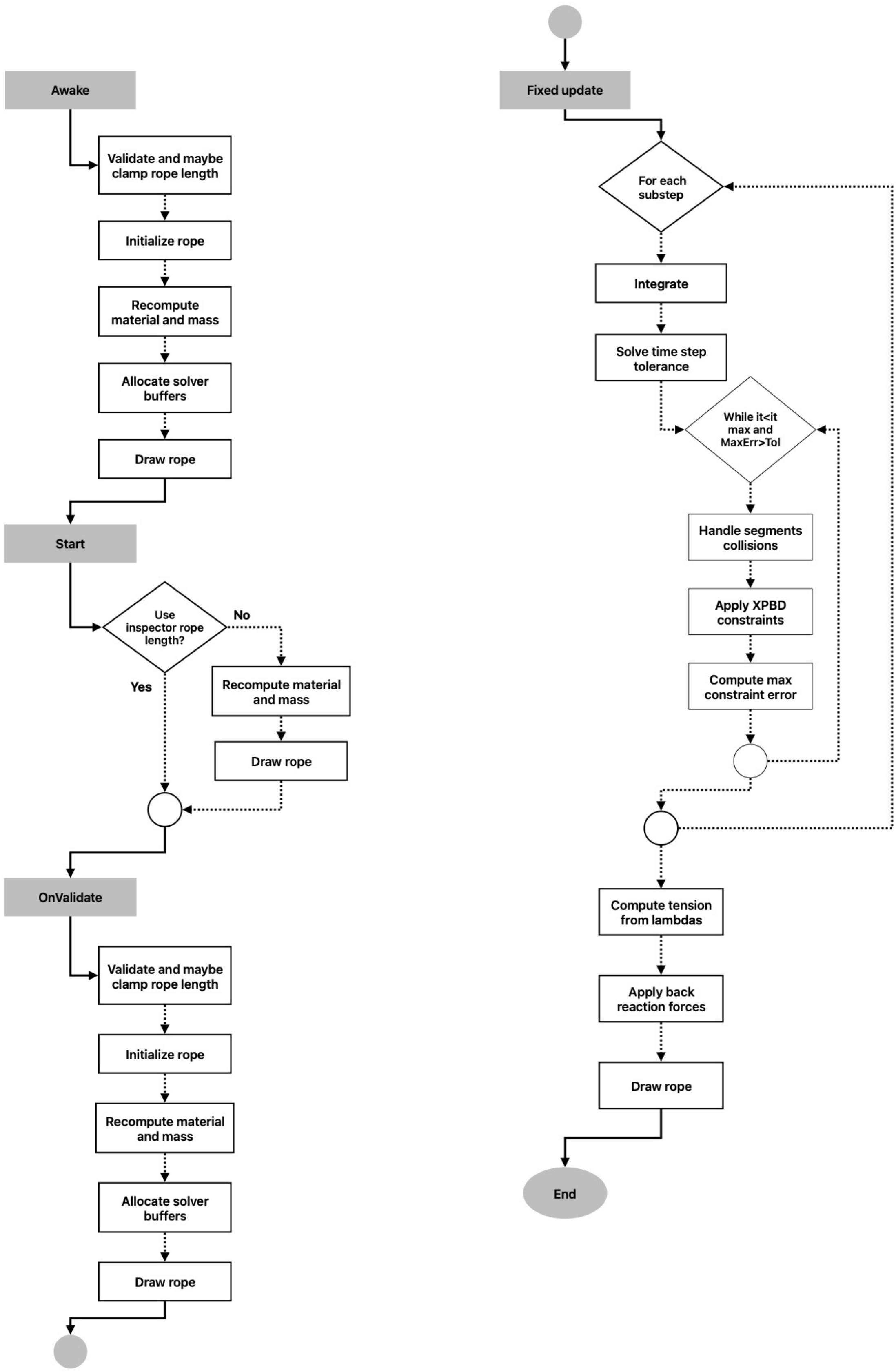


Figure 5.1: VerletTether.cs flowchart

5.2 Description of the implemented functions

5.2.1 Awake( )

The Awake() function is invoked automatically by the Unity engine during the initialization phase of the simulation, before the first frame update and prior to the execution of the Start() coroutine. Its primary role is to establish all the fundamental elements required for the tether model. More specifically, the function performs the following tasks:

1. Parameter validation – the method calls ValidateAndMaybeClampRopeLength()
2. Geometrical initialization – through InitializeRope()
3. Physical parameter computation – by invoking RecomputeMaterialAndMass()
4. Solver buffer allocation - through AllocateSolverBuffers()
5. Initial visualization – by invoking DrawRope()

5.2.2 Start( )

The Start() function in Unity is executed once, immediately after Awake() and before the first simulation step. In this implementation it is defined as a coroutine, which allows the initialization to be distributed across multiple frames, thereby ensuring that all Unity components (such as Rigidbody objects and colliders) are fully instantiated before the tether begins its dynamic evolution.
The function operates as follows:

1. Rope length adjustment (conditional) - if the flag useInspectorRopeLength is disabled, the function recalculates the rope length directly from the geometric distance between the anchor points
2. Recomputation of physical properties - through RecomputeMaterialAndMass()
3. Visualization - to refresh the initial graphical representation of the tether DrawRope() is called

5.2.3 OnValidate( )

The OnValidate() function is a special Unity callback that is automatically invoked in the Editor environment whenever one of the serialized fields of the script is modified through the Inspector panel. Unlike Awake() and Start(), which are executed during the runtime phase, OnValidate() operates at design time and serves as a tool for maintaining internal consistency of the simulation model while the user configures the tether parameters. As the preceding functions, it performs the following tasks:

1. Parameter verification and clamping – it calls ValidateAndMaybeClampRopeLength()
2. Geometry reconstruction – it invokes InitializeRope()
3. Update of physical properties – through RecomputeMaterialAndMass()
4. Solver data reallocation – it calls AllocateSolverBuffers()
5. Graphical refresh – finally, DrawRope() is executed

5.2.4 ValidateAndMaybeClampTheRope( )

The function is designed to guarantee the physical feasibility of the tether's initial configuration by checking the consistency between the user-defined rope length and the spatial arrangement of the anchor points.

In practice, the function computes the geometric distance between the two rigid bodies (the start and end anchors of the tether). It then compares this value with the desired rope length specified in the Inspector or computed at runtime. Since a flexible tether cannot be shorter than the straight-line distance between its endpoints, if the ClampRopeLengthToAnchors option is enabled, the function enforces the following condition:

$$L \geq \|x_{end} - x_{start}\|$$

where L is the prescribed rope length, and x_{start}, x_{end} denote the positions of the two anchor points. If the assigned rope length violates this inequality, the function automatically clamps it to the minimum admissible value:

$$L \leftarrow \|x_{end} - x_{start}\|$$

5.2.5 InitializeRope( )

The InitializeRope() function is responsible for constructing the discretized representation of the tether at the beginning of the simulation. Since the physical tether is modeled as a flexible structure, it cannot be represented as a single rigid element; instead, it is approximated by a sequence of point masses (nodes) connected by segments that enforce distance constraints.
The function executes the following key steps:

1. Segment length calculation: calculates the length of a segment by dividing the tether length by the number of segments chosen for its discretization.
2. Node generation: the positions of these nodes are initialized by interpolating linearly between the two anchor points (startRB and endRB). This ensures that the tether is created in a straight configuration, with equally spaced nodes. The two anchor points are set as locked though a flag.
3. Initial conditions: the current and previous positions of each node are set. This setup ensures that the Verlet integration scheme can accurately calculate velocities and accelerations from the first simulation step.

5.2.6 RecomputeMaterialAndMass( )

This function is responsible for updating all the derived physical parameters of the tether whenever its geometric or material properties change.

It computes the following values:

1. Linear density: $\mu = A \cdot \rho$
2. Axial stiffness for each segment: $k = \frac{EA}{L_i^0}$
3. Constraint compliance (XPBD): $\tilde{\alpha}_i = \frac{1}{k \cdot s^2}$
4. Mass distribution along the nodes: $m_i = \mu \cdot L_i^0$

5. Inverse mass of each point:

$$w_i = \begin{cases} \frac{1}{m_i} & 1 \leq i \leq N-1 \\ 0 & i = 0 \text{ or } i = N \end{cases}$$

Where:

$$L_i^0 = \text{segment i rest length}$$
$$s = \text{substep value [s]}$$

5.2.7 AllocateSolverBuffers( )

The AllocateSolverBuffers() function serves as a preparatory routine, initializing the solver's internal buffer arrays necessary for the constraint resolution phase of the XPBD method. Specifically, it allocates and resets the arrays responsible for storing Lagrange multipliers and the tension values associated with each segment.

5.2.8 DrawRope( )

The function is responsible for the visualization of the discretized rope in the simulation environment. It iterates over the tether points and renders line primitives between adjacent nodes. By doing so, it provides a real-time graphical representation of the rope's configuration, deformation, and motion as computed by the solver.

5.2.9 FixedUpdate( )

This function represents the main loop of the code. In order to obtain a finer temporal integration and enforce constraints, the time step dt is decomposed into S substeps.

5.2.10 Integrate( )

The funtcion integrates the internal points with the Verlet integration

$$v = (x_i - x_{i-1}) \cdot \text{damping}$$
$$x_i = x_i + v + a \cdot \Delta t^2$$

In the numerical formulation, a damping term was incorporated to ensure energy dissipation and to mitigate undesired oscillations during the rope dynamics simulation.

5.2.11 SolveTimeStep_Tolerance( )

This function makes sure the distance between consecutive rope segments stays within the set limits during each step of the simulation. It follows an iterative method to address the XPBD constraints until the deviation is small enough or a maximum number of iterations is reached. This way, it fixes any deviations that may have happened.

5.2.12 HandleSegmentCollisions( )

This function enforces non-penetration constraints between each tether segment and the colliders present in the environment. Its implementation employs a simplified yet computationally efficient strategy: for each segment, a single virtual bounding sphere is constructed, centered at the geometric midpoint of the segment and with a radius equal to half the segment's length. This ensures that the entire segment is enclosed within the sphere.

The Unity physics engine uses the OverlapSphere operation to find all colliders that intersect the bounding sphere. For each collider it finds, the algorithm calculates the closest point on the collider's surface to the segment's midpoint. Then, it checks if the midpoint is in the penetration region of the bounding sphere. If there is penetration, the algorithm calculates the penetration depth and creates a correction vector that goes from the collider surface to the segment's midpoint.

This correction applies equally to the two endpoints of the segment, unless one of the endpoints is constrained, like being locked to an anchor. This way, the method pushes the tether outward from the obstacle, keeping physical realism intact.

5.2.13 ApplyXPBDConstraints( )

The function keeps the tether from stretching by using the Extended Position-Based Dynamics (XPBD) method on each segment.

For each pair of consecutive points, the algorithm calculates the current constraint violation. This calculation is the difference between the actual segment length and its set rest length. The violation then goes through the XPBD update rule, which adds a compliance term. This term helps control the effective stiffness of the constraint while ensuring numerical stability. The method also tracks a Lagrange multiplier for each constraint and updates it at every iteration.

The correction is shared between the two endpoints of the segment based on their inverse masses. Nodes with higher mobility get larger displacements. In contrast, anchored or heavily constrained nodes stay nearly fixed. As a result, the algorithm adjusts the tether configuration to reduce constraint violation while maintaining physical realism and considering material compliance.

By iterating this process across all segments, ApplyXPBDConstraints() maintains the rope's rest length distribution, thereby ensuring that the tether neither stretches nor compresses unrealistically under dynamic loading. In formulas, for each pair of points, the following quantities are defined:

$$\begin{aligned}
\lambda &= \text{Lagrange multiplier} \\
\text{Compliance} \;\; \alpha &= \frac{1}{k} \\
\tilde{\alpha} &= \frac{\alpha}{\Delta t^2} \\
\bar{\Delta} &= \bar{x}_B - \bar{x}_A \\
\text{Constraint violation} \;\; C &= \bar{\Delta} - L_{Segment} \\
\text{Direction} \;\; \hat{n} &= \frac{\bar{\Delta}}{\|\bar{\Delta}\|} \\
\text{Effective mass of the point A} \;\; w_A &= \frac{1}{m_A} \\
\text{Effective mass of the point B} \;\; w_B &= \frac{1}{m_B} \\
w_{sum} &= w_A + w_B
\end{aligned}$$

Then, the correction is computed according to the following expressions:

$$\Delta\lambda = -\frac{C + \tilde{\alpha}\lambda}{w_{sum} + \tilde{\alpha}}$$
$$\lambda = \lambda + \Delta\lambda$$
$$\bar{x}_A = \bar{x}_A + w_A \Delta\lambda \cdot \hat{n}$$
$$\bar{x}_B = \bar{x}_B - w_B \Delta\lambda \cdot \hat{n}$$

5.2.14 ComputeMaxConstraintError( )

For each segment connecting two consecutive nodes, the function computes the absolute difference between the current segment length and its prescribed rest length.

Among all segments, the maximum deviation is chosen as the global constraint error. This value has two roles: first, it acts as a diagnostic tool to measure the accuracy of constraint enforcement at a specific iteration. Second, it sets the stopping rule for the iterative solver. During each physics substep, the constraint solver (SolveTimeStep_Tolerance) makes corrections repeatedly until either the maximum error drops below a set tolerance level or the maximum number of iterations is reached.

In this way, the functions acts as a convergence monitor, ensuring that the constraint projection process does not terminate prematurely while also avoiding unnecessary iterations once the tether geometry has been corrected within the desired numerical tolerance.

$$C = \|\bar{x}_B - \bar{x}_A\| - L_{\text{segment}}$$

Evaluation of the maximum absolute value |C| over the rope:

$$C_{\max} = \max_{(A,B)} \mid \|\bar{x}_B - \bar{x}_A\| - L_{\text{segment}} \mid$$

5.2.15 ComputeTensionFromLambdas( )

The function evaluates the tensile force acting along each rope segment by exploiting the values of the Lagrange multipliers obtained during the constraint resolution process. Specifically, once the constraint violation C and the corresponding correction multiplier $\Delta\lambda$ have been determined, the function computes the internal tension as a function of the updated multipliers, thereby providing a consistent measure of the forces transmitted through the rope:

$$T_i = -\frac{\lambda_i}{\Delta t^2} \tag{5.1}$$

A non-physical situation may arise when the computed value of a segment tension T_i becomes negative, which would correspond to compressive forces that a real tether cannot sustain. To prevent such artifacts, the implementation explicitly enforces $T_i = 0 \text{ if } T_i < 0$.

5.2.16 ApplyBackReactionForces( )

This function ensures that the interaction between the tether and its anchor points is dynamically consistent with Newton's third law of motion. Tensile forces are computed in ComputeTensionFromLambdas( ), and the equal and opposite reactions of these forces must be applied to the rigid bodies to which the tether is attached (e.g., the floater or the spool). The function achieves this by evaluating the net tensile force exerted by the first and last tether segments on the two anchor points, and then applying the corresponding opposite forces to the rigid bodies via Unity's physics interface. This mechanism allows the anchors to respond realistically to the tether tension.

5.3 PID tension controller for deployed tether

The PID tension controller regulates the tension acting along the discretized tether by commanding corrective roll and pitch inputs to the supporting platform (the granite bench), thereby inducing controlled accelerations in the floater. As described in the chapter concerning the scaling factors, the control acceleration is computed by the PID controller for the real system, and only immediately before converting the acceleration into the rotation angles of the granite bench is it transformed into the "table system", through equation (3.2).
The control chart is presented below:

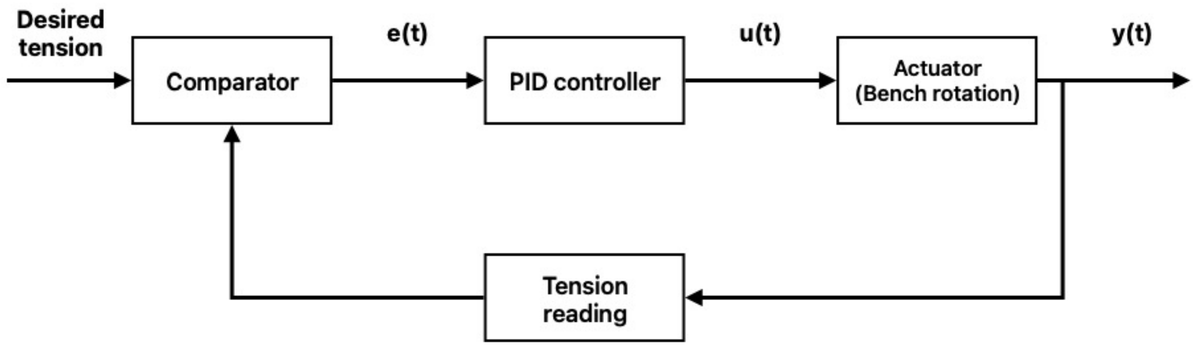


Figure 5.2: PID tension control diagram, deployed tether

The primary objective is to regulate the tether tension T to a desired tension T_d. However, because derivative action on raw tension can amplify measurement noise, the measured tension is first passed through a first-order exponential low-pass filter with user-selected cutoff frequency:

$$\tilde{T}_{k+1} = \tilde{T}_k + (1-\alpha)(T_k - \tilde{T}_k)$$

Where $\alpha = e^{-2\pi f_c \Delta t}$. In addition, for the first iteration $\tilde{T}_0 = T_0$
After filtering the measured tension, the different errors are calculated:

$$e = T_d - \tilde{T}$$
$$e_i = \int e\,dt = e \cdot dt$$
$$e_d = \frac{de}{dt}$$

And the standard PID law is used:

$$u = K_p\,e + K_i \int e\,dt + K_d\,\frac{de}{dt} = K_p\,e + K_i\,e_i + K_d\,e_d \tag{5.2}$$

Where u = control acceleration
Once the control acceleration has been calculated, the direction in which it have to be applied is then computed:

$$\bar{d} = \bar{x}_e - \bar{x}_s$$
$$\hat{n} = \frac{\bar{d}}{\|\bar{d}\|}$$

And finally:

$$\bar{u} = u \cdot \hat{n}$$

The acceleration in the real system is converted into the acceleration in the table reference frame:

$$\bar{u}_{bench} = \bar{u}_{real} \cdot \frac{\lambda_t^2}{\lambda_L}$$

The maximum tilt angle of the bench imposed on the actuators is 2°; it is therefore reasonable to assume the hypothesis $sin\theta \approx \theta$, in order to convert the control acceleration in angles.

$$\phi_{des} = \frac{\bar{u} \cdot \hat{k}}{g} \quad \text{Roll angle}$$

$$\theta_{des} = -\frac{\bar{u} \cdot \hat{i}}{g} \quad \text{Pitch angle}$$

The motion of the actuators, and consequently the inclination of the bench, is not instantaneous. To simulate their inertia, the SmoothDampAngle function, already available in the Mathf library, is employed; this function enables the desired angle to be reached from the current one within a specified time interval.

5.3.1 Results of PID-Based tension control for a fully deployed tether

To assess the effectiveness of this algorithm, it was initially imperative to calibrate the values of the three gains. A trial-and-error methodology was employed until convergence to the desired tension level was achieved.
The calibration process resulted in the following parameter values:

$$K_p = 1$$
$$K_i = 0.001$$
$$K_d = 0.81$$

The test was carried out using the following data:

$$T_{des} = 0.08\ N$$
$$\bar{p}_{start} = (0,\ 0.9,\ 0)$$
$$f_{cutoff} = 3\ Hz$$
$$\beta_{max} = 2^\circ$$
$$t_{actuators} = 2\ s$$

Where β is the bench tilt angle.

The results reported in figure 5.3 confirm that the controller is able to maintain the commanded tension with acceptable overshoot and settling time (approximately 1–2 seconds). In particular, the evolution of the X-position remains constant throughout the experiment, indicating that the control action does not induce unwanted motion and that the geometry of the setup preserves symmetry in the longitudinal direction. The oscillations along the y-axis are a result of the fact that each time the bench tilts, it either elevates or depresses the center of mass of the floater. In contrast, the Z-position exhibits a rapid settling toward a stable equilibrium near 0.2 m, with a transient that is well damped and free from overshoot.

The tension trace itself provides the clearest indication of control performance. Starting from zero, the tension increases smoothly and reaches the target value within approximately two seconds, exhibiting minimal overshoot and settling quickly to a nearly constant level with only small residual oscillations. The mean tension over the free segments remains very close to 0.08 N, confirming that the controller maintains the desired pre-load throughout the test.

In the post-deployment phase, tension control is essential to preserve a stable relative distance and to mitigate vibrations induced by orbital perturbations or control actions. The PID law can adjust small attitude or thrust corrections to maintain a constant preload, preventing the tether from entering slack conditions that would result in loss of controllability and unwanted impacts when tension is suddenly restored. Moreover, accurate tension regulation supports attitude stabilization: since tether tension directly affects the torque transmitted to the deputy, controlling it can indirectly stabilize its relative orientation or spin rate.

From a broader mission perspective, tension control contributes to the robustness and safety of the system. By limiting peak forces and ensuring smooth tension variations, the controller reduces mechanical fatigue and minimizes the risk of structural damage or overstress in the tether material. In electrodynamic tether missions, where current flow depends on tether geometry and tension, maintaining a constant force also ensures consistent electrical and thermal behavior.

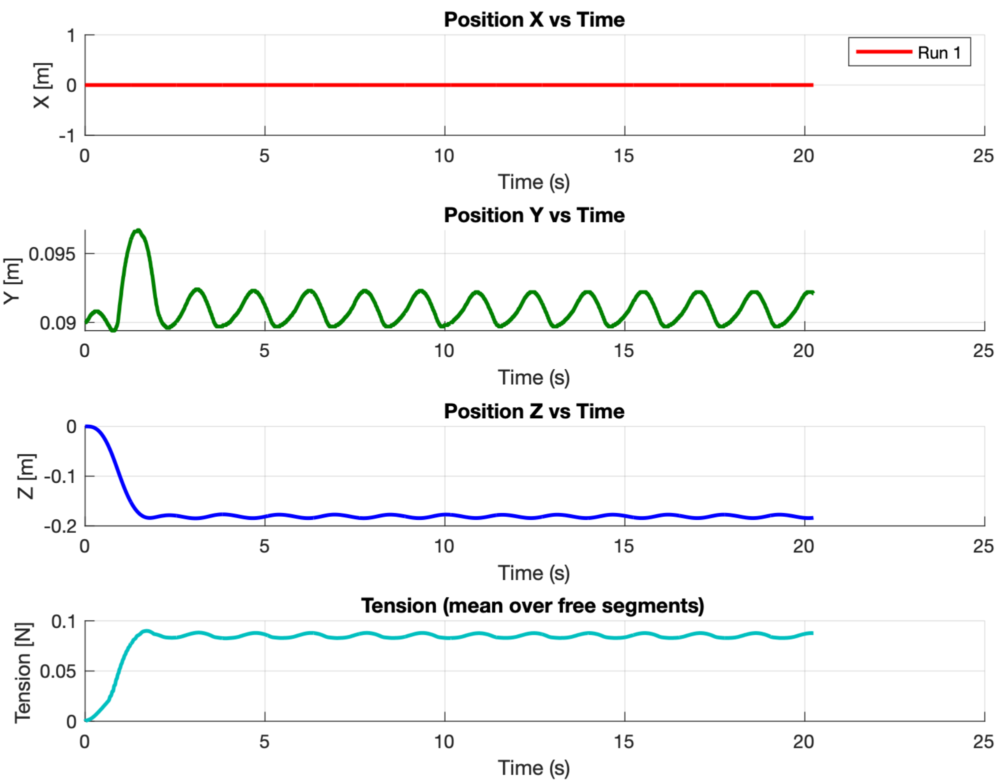


Figure 5.3: Deployed case - PID tension results

5.4 PID position controller for deployed tether

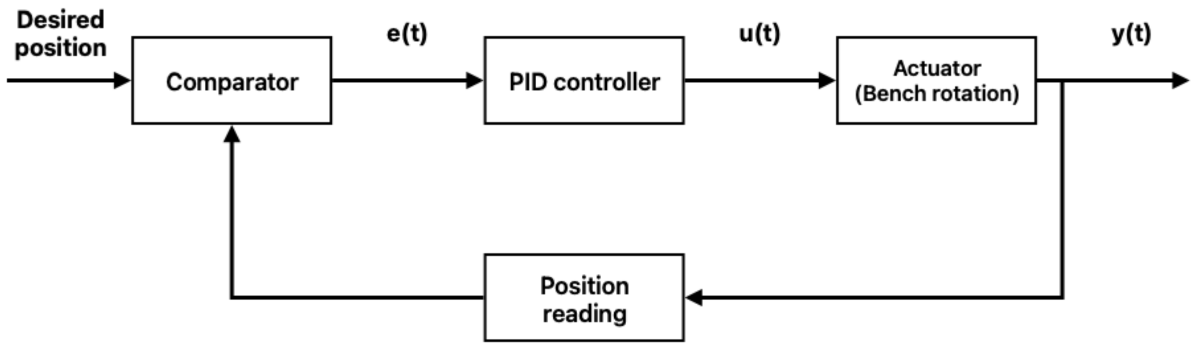


Figure 5.4: PID position control diagram, deployed tether

The operation of the PID position controller is entirely analogous to that of the tension controller, with the sole difference that the error is computed with respect to the desired position (specified by the user) rather than the tension. Similarly to the previous case, this controller also determines the control acceleration within the "real system", and only immediately before converting the acceleration into angular values is it transformed into the "table system".

$$\bar{e} = \bar{p}_{des} - \bar{p}$$
$$\bar{e}_i = \int \bar{e}\, dt = \bar{e} \cdot dt$$
$$\bar{e}_d = \frac{d\bar{e}}{dt} = \frac{\bar{e}}{dt}$$

Standard PID law:

$$\bar{u} = K_p\, \bar{e} + K_i \int \bar{e}\, dt + K_d \frac{d\bar{e}}{dt} = K_p\, \bar{e} + K_i\, \bar{e}_i + K_d\, \bar{e}_d \tag{5.3}$$

Conversion of the control acceleration into the "table system":

$$\bar{u}_{bench} = \bar{u}_{real} \cdot \frac{\lambda_t^2}{\lambda_L}$$

Transformation of the control acceleration into angles:

$$\phi_{des} = \frac{\bar{u} \cdot \hat{k}}{g} \quad \text{Roll angle}$$
$$\theta_{des} = -\frac{\bar{u} \cdot \hat{i}}{g} \quad \text{Pitch angle}$$

In this case as well, the SmoothDampAngle function is used to simulate the actuation inertia.

5.4.1 Results of PID-Based position control for a fully deployed tether

In this test, the calibration process led to the following values:

$$K_p = 2.4$$
$$K_i = 0$$
$$K_d = 1.8$$

The test was carried out using the following data:

$$\bar{p}_{des} = (-5,\ 0.9,\ -10)$$
$$\bar{p}_{start} = (0,\ 0.9,\ 0)$$
$$\beta_{max} = 2^\circ$$
$$t_{actuators} = 2\ s$$

Where $\bar{p}_{des}$ is express with respect to the real system. The following results were obtained:

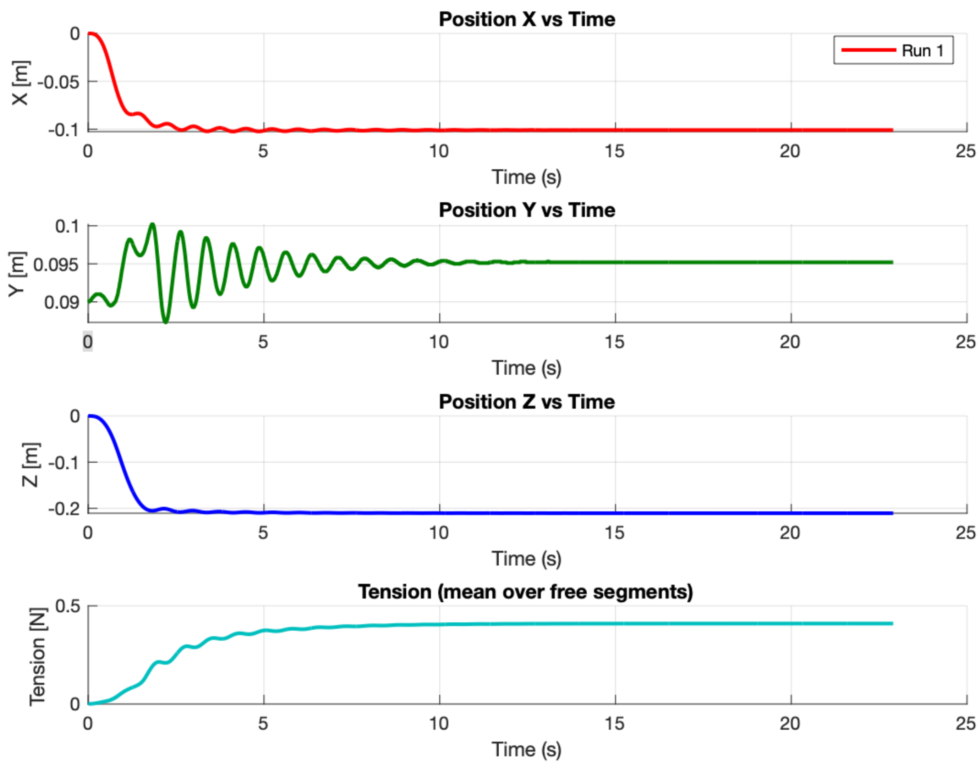


Figure 5.5: Deployed case - PID position results

The results of the PID position control reveal the dynamic response of the system along the three spatial axes, as well as the corresponding tether tension profile. In the X-direction, the platform exhibits a rapid convergence towards the equilibrium configuration, with only small oscillatory components before stabilizing around the desired position.
In contrast, the Y-direction response is characterized by a more pronounced oscillatory behavior, but, as mentioned, the controller does not manage this direction.
Along the Z-direction, the graph indicates a quick movement toward the target position. This is followed by a small overshoot and then stabilization.

The evolution of the tether tension reflects the effectiveness of the position control scheme. The tension rises smoothly during the initial transient phase, avoiding abrupt peaks that could compromise system stability, and progressively converges to a steady-state value consistent with the imposed equilibrium configuration. The absence of significant fluctuations in the steady state indicates that the PID law provides a consistent balance between position regulation and tension stabilization.

In conclusion, the results show that the PID controller effectively directs the system to the desired position while keeping the tether tension stable.

Position control finds numerous applications. During the post-deployment phase, once the tether reaches its full length, position control ensures that the deputy satellite remains in a quasi-stationary position relative to the chief. This is crucial for missions involving formation flying, rendezvous and docking simulations. In these scenarios, maintaining a stable relative geometry is essential to ensure that the measurements taken by the two satellites remain coherent and that the tether does not introduce uncontrolled perturbations. Additionally, position control allows the deputy to return to its desired configuration after external disturbances, such as orbital perturbations, micro-impacts, or transient attitude motions. This ensures that the nominal relative trajectory is quickly re-established without residual oscillations or long-term drift.

Furthermore, position control can aid in attitude stabilization and station keeping. Since the tether tension acts along the line connecting the two spacecraft, small position corrections generate torques that can be utilized for attitude control without the need for continuous propellant expenditure.

5.5 LQR position controller for deployed tether

The LQR position controller implements a Linear Quadratic Regulator (LQR) to achieve precise position control of the tethered floater on the granite bench.
The control flow chart is presented below:

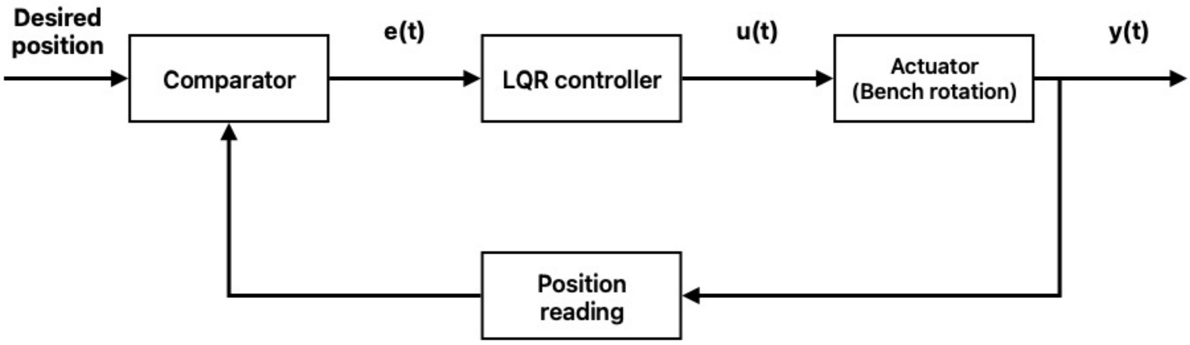


Figure 5.6: LQR position control diagram, deployed tether

The design of the LQR controller requires a state-space representation of the relative orbital dynamics. In the local–vertical local–horizontal (LVLH) reference frame, the motion of a deputy satellite with respect to a chief in a circular reference orbit of mean motion ω can be described by the Hill–Clohessy–Wiltshire (HCW) equations. Considering only the in-plane dynamics (radial x and tangential y coordinates), these equations read:

$$\begin{cases} \ddot{x} - 2\,\omega\,\dot{z} - 3\omega^2\,x = \frac{F_x}{m} \\ \ddot{z} + 2\,\omega\,\dot{x} = \frac{F_z}{m} \end{cases} \quad \begin{cases} \ddot{x} - 2\,\omega\,\dot{z} - 3\omega^2\,x = u_x \\ \ddot{z} + 2\,\omega\,\dot{x} = u_z \end{cases} \tag{5.4}$$

Where:

- x and z are the radial and along-track relative position
- $\dot{x}$ and $\dot{z}$ are the corresponding velocities
- $u_x,\ u_z$ are the control accelerations in the radial and tangential directions, respectively
- $\sqrt{\frac{\mu}{a^3}}$ is the orbital angular velocity of the chief

The HCW equations can be written as a linear time-invariant system:

$$\dot{x} = A\,x + B\,u$$

where:

$$\bar{\bar{A}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 3\omega^2 & 0 & 0 & 2\omega \\ 0 & 0 & -2\omega & 0 \end{bmatrix}, \qquad \bar{\bar{B}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

From the diagonal value inserted by the users, the matrices Q and R can be built, which are necessary to solve Riccati equation. Q penalizes deviations from the desired relative trajectory, and R penalizes control effort. Having created the two matrices, the matrix P can be calculated,

as the unique positive semi-definite solution of continuous algebraic Riccati equation.
At this stage, the gain matrix K is computed: $K = R^{-1}B^TP$

The error/state vector is introduced:

$$\bar{e} = \begin{bmatrix} x_{des} - x \\ z_{des} - z \\ \dot{x} \\ \dot{z} \end{bmatrix}$$

Finally, the control acceleration vector can be calculated:

$$\bar{u}(t) = -\bar{\bar{K}} \cdot \bar{\mathbf{x}}(t)$$

The LQR controller operates in the "real system"; therefore, before converting the control acceleration into angles, it must first be referred to the "table system" through equation (3.2):

$$\bar{u}_{bench} = \bar{u}_{real} \cdot \frac{\lambda_t^2}{\lambda_L}$$

After having converted the control acceleration, the conversion into angles can be carried out:

$$\phi_{des} = \frac{\bar{u} \cdot \hat{k}}{g} \quad \text{Roll angle}$$

$$\theta_{des} = -\frac{\bar{u} \cdot \hat{i}}{g} \quad \text{Pitch angle}$$

To simulate the actuation inertia, the SmoothDampAngle function is used.

5.5.1 Results of LQR-Based position control for a fully deployed tether

In the case of position control based on the LQR formulation, a trial-and-error procedure was likewise required in order to determine the diagonal entries of the weighting matrices Q and R that yield the most satisfactory regulation of the system. The selection of these parameters plays a crucial role, as it directly reflects the compromise between state accuracy and control effort. As a preliminary step, the tuning was intentionally biased towards relatively large values of Q_{11} and Q_{22}, thereby assigning a dominant weight to the position states. This design choice was motivated by the specific objective of enforcing a highly accurate position tracking, even at the cost of allowing slightly higher control effort. The subsequent refinements of the remaining parameters were then carried out to balance overall stability and actuation smoothness, leading to the final configuration reported below.

$$\begin{aligned} Q_{11} &= 3200 & & \\ Q_{22} &= 3200 & \qquad R_{11} &= 140 \\ Q_{33} &= 200 & \qquad R_{22} &= 140 \\ Q_{44} &= 200 & & \end{aligned}$$

The test was conducted using the following data:

$$\bar{p}_{des} = (-5,\ 0.9,\ -10)$$
$$\bar{p}_{start} = (0,\ 0.9,\ 0)$$
$$\beta_{max} = 2^\circ$$
$$t_{actuators} = 2\ s$$
$$\Omega = 0.1\ \frac{rad}{s}$$

Where $\bar{p}_{des}$ is express with respect to the real system, while Ω with respect to the table system.

The results are reported in the following image:

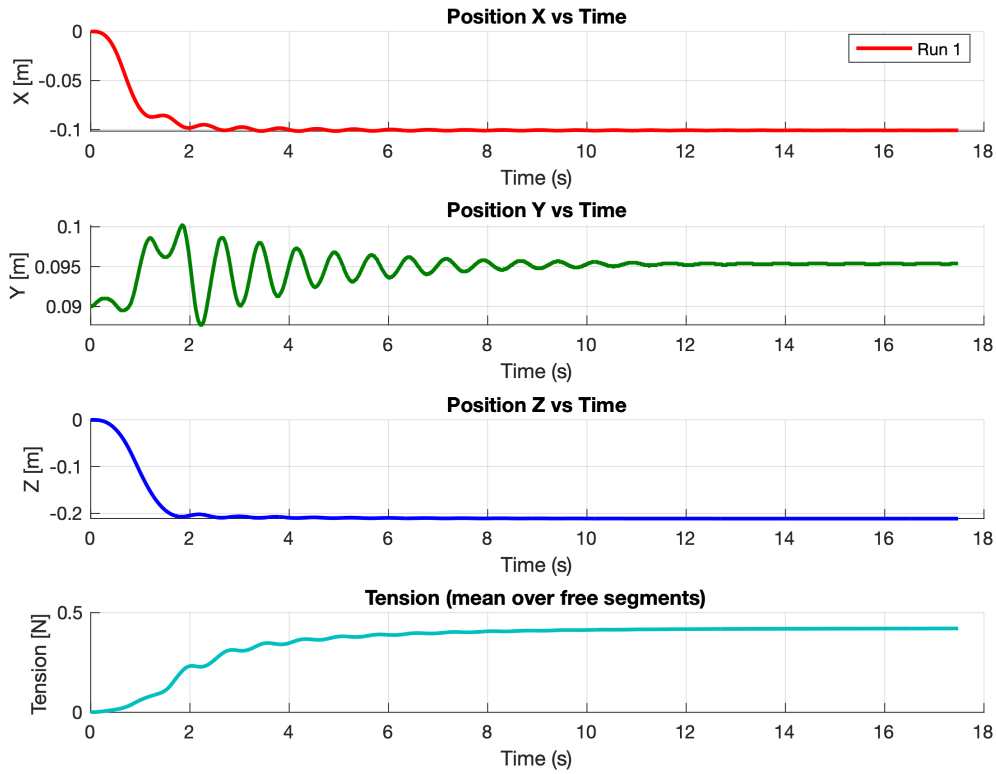


Figure 5.7: Deployed case - LQR position results

Along the X-axis, the system rapidly converges to the desired equilibrium with minimal overshoot and an almost critically damped response. The absence of sustained oscillations indicates that the optimal feedback gains computed through the LQR formulation effectively suppress deviations, providing both fast and smooth stabilization.

The Z-axis response shows a quick and controlled approach to the target position, with nearly no overshoot. As a result, the system reaches stability in under three seconds.

The tension gradually rises to its equilibrium value. It reaches a steady state smoothly and without overshooting.

Chapter 6

Software implementation of the deployment case

This chapter presents the implementation of the tether deployment on the granite bench, employing a PID-based tension control scheme and halting the deployment process once the desired deployed tether length is reached. The model aims to reproduce, at a reduced scale, the optimal architecture for a tethered satellite system derived from the analyses discussed in the previous chapters. The rotating spool system is modeled as a spool around which the tether is wound in a helical configuration. The tether itself is represented with the material properties of a cotton thread (as in the deployed case), using the XPBD method in combination with Verlet integration. Since many functions are shared with the code developed for the deployed-tether case, this chapter will focus in detail only on the newly introduced functions.

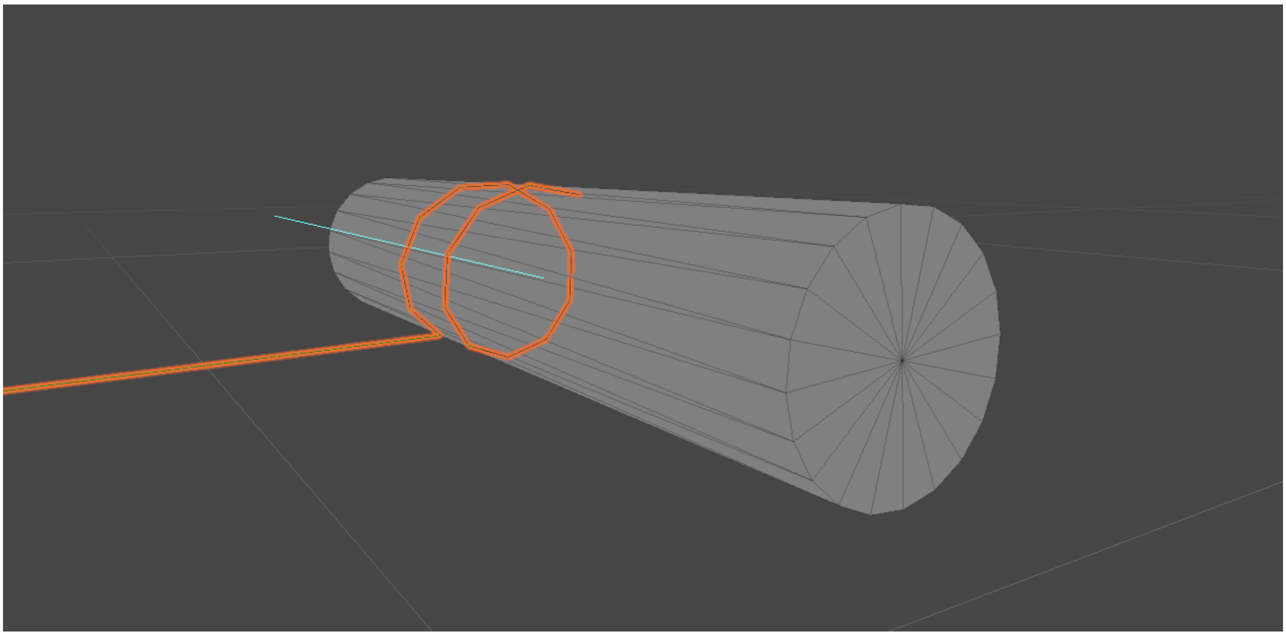


Figure 6.1: Spool in Unity

6.1 Deployment implementation - XPBDTetherWithSpool.cs

This code (whose flowchart is shown in figure 6.2) implements tether deployment via a rotating spool. It maintains two anchors: the spool side and the floater side, and it applies back-reaction forces to the floater.
The spool is represented kinematically by a cylinder with a user-defined radius and helical pitch. A subset of the rope's discrete points is constrained to lie on a helical path around the spool. These "wrapped" points advance along the helix as the spool rotates. Deployment occurs by releasing the outermost wrapped point when its radial direction enters a configurable angular window around the lowest point, simulating gravity-assisted unwinding. The first point is permanently attached to the spool to model a fixed take-off. Upon deployment completion, the script suspends the spool's motion. Furthermore, the script continuously estimates segments tensions from the XPBD multipliers.

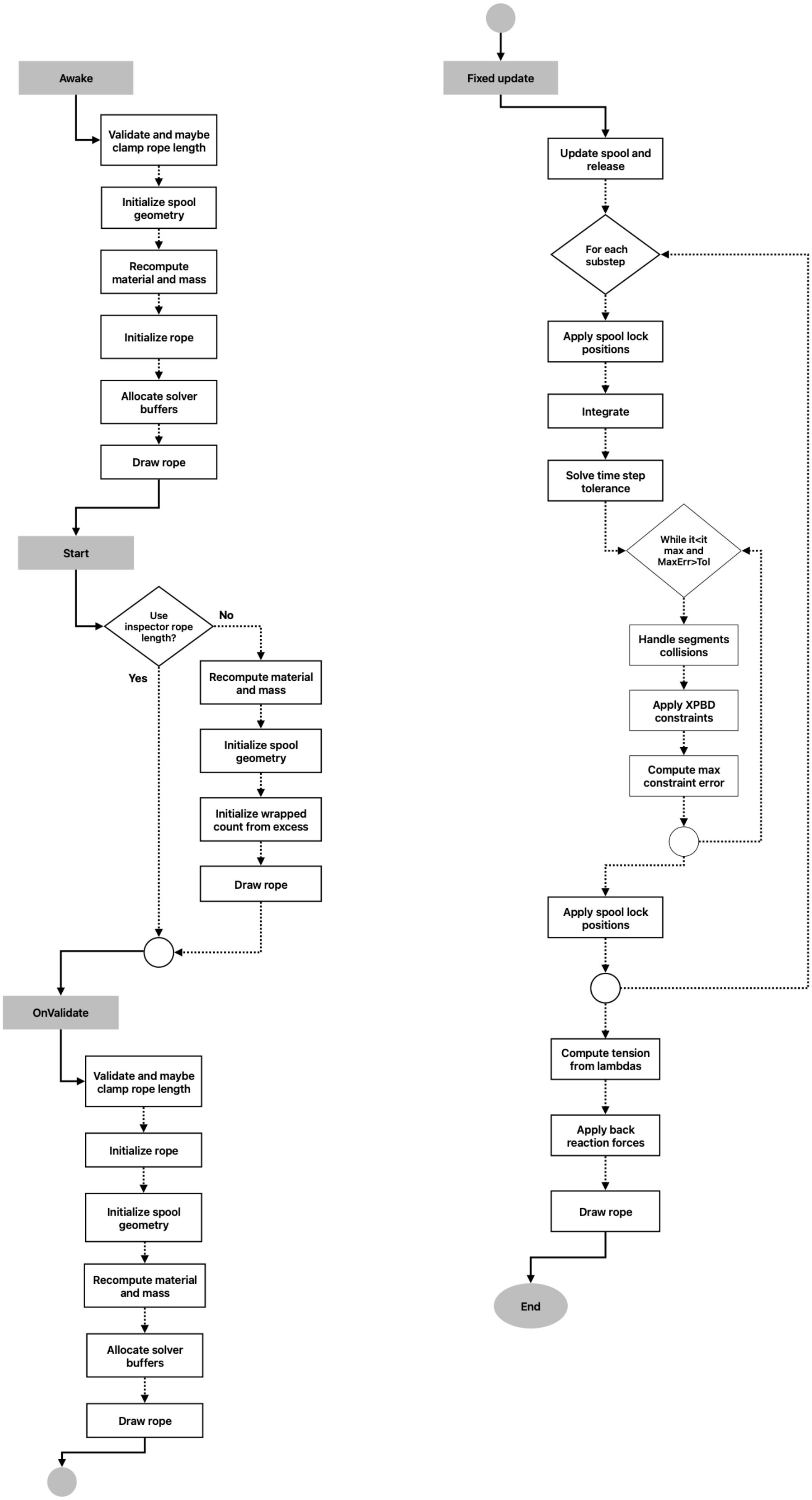


Figure 6.2: XPBDTetherWithSpool.cs flowchart

6.2 Desctiption of new implemented functions

6.2.1 InitializeSpoolGeometry( )

The InitializeSpoolGeometry() method is responsible for constructing the helical geometry that governs the tether's adhesion to and unwinding from the spool. It also identifies the physical reference directions necessary for the deployment logic.

The function begins by establishing the helical axes and the radial reference direction. Subsequently, the helix turn length is computed.

$$L_{helix} = \sqrt{(2\pi R)^2 + p^2} \qquad \text{where} \quad R = \text{spool radius} \quad \textit{and} \quad p = pitch \tag{6.1}$$

In order to determines the direction of the lowest point of the spool, in which segments switch from being constrained to being free, the global downward vector $\bar{g} = (0, -1, 0)$ is projected onto the plane orthogonal to the spool axis. Finally, the method invokes InitializeWrappedCountFromExcess( ), in order to ensure that the number of points initially considered adherent to the spool is consistent with the available slack in the tether.

6.2.2 InitializeWrappedCountFromExcess( )

This function determines the number of tether segments that are wound on the spool at the beginning of the simulation. This is essential for ensuring that the simulated tether starts in a physically feasible state, with only the excess length beyond the anchor distance (with a slight offset to prevent pre-tensioning of the tether) being available for deployment.

The first step is to evaluate the distance between the two anchor points:

$$d = \|\bar{p}_{start} - \bar{p}_{end}\|$$

Then, the function calculates the excess length subtracting the distance between the anchors and an additional safety margin from the total rope length:

$$L_{excess} = max(0,\ L_{rope} - d - \Delta s_{slack})$$

In the end, it is possible to express this excess in terms of discrete segments:

$$N_{excess} = \frac{L_{excess}}{L_{seg}}$$

To avoid degenerate conditions, the result is clamped between 0 and N-1, where N is the total number of the segments. If the computed value falls below the minimal required threshold, the function forces the value to that minimum value.

6.2.3 HelixPosition( )

The function HelixPosition( ) computes the spatial position of tether points constrained to lie on the surface of the spool. The input parameter $s_{along\,helix}$ represents the arclength coordinate along the elix (distance traveled along the tether), θ_{offset} is an additional angular offset that accounts for the instantaneous rotation of the spool, and L_{helix} is the length of a winding.

The procedures begins by computing the angular position of the tether point along the helix. Given the helix turn length L_{helix}, the angle is determined as:

$$\theta = \frac{s_{along\,helix}}{L_{helix}} \cdot 2\pi + \theta_{offset}$$

Knowing the spool axis $\bar{a}$ and the radial vector $\bar{u}$, the second orthogonal direction is:

$$\hat{v} = \frac{\bar{a} \times \bar{u}}{\|\bar{a} \times \bar{u}\|}$$

Thus, the radial and the axial displacements can be calculated:

$$\bar{r}(\theta) = R(\bar{u} \cdot cos\theta + \bar{v} \cdot sin\theta)$$
$$\bar{z}(s_{along\,helix}) = \bar{a} \ \cdot \ \frac{s_{along\,helix}}{L_{helix}} p$$

where p is the helical pitch.

In the end, the total position of the tether point is reconstructed as the sum of the spool's global position $\bar{x}_0$ with the radial and axial components:

$$\bar{x}(s_{along\,helix}, \theta_{offset}) = \bar{x}_0 + \bar{r}(\theta) + \bar{z}(s_{along\,helix})$$

6.2.4 HelixRadialDir( )

The function HelixRadialDir() computes the direction associated with a tether point at a given arclength along the helix. This direction vector is critical for the release logic: it is compared with the "bottom direction" determined by gravity in order to decide when a segment of the tether should detach from the spool.
The method calculates the angular parameter in the same way as in HelixPosition, and reuses the same othonormal basis vectors.

The point direction is given by:

$$\bar{n}_r(\theta) = \frac{\bar{u} \cdot cos\theta + \bar{v} \cdot sin\theta}{\|\bar{u} \cdot cos\theta + \bar{v} \cdot sin\theta\|}$$

This vector points radially outward from the spool center.

6.2.5 ApplySpoolLockPositions( )

This function enforces the kinematic adhesion of the wound portion of the tether to the spool surface at the beginning and end of each physics substep, so that, during constraint projection and collision handling, these points do not drift off the cylinder.

For each wound index $i \in \{0, ..., N_{wrap} - 1\}$ the code computes the arclength along the helix:

$$s_i = i \ \cdot \ L_{seg}$$

in addition, it evaluates the point position on the helix through the HelixPosition function: $\bar{x}_i = HelixPosition(s_i, \theta_{offset})$ and subsequently updates the position and records the previous one. Finally, the function imposes $m^{-1} = 0$, which is the inverse mass used by the XPBD solver: this value effectively removes the point from the dynamical solve.

6.2.6 UpdateSpoolAndRelease( )

UpdateSpoolAndRelease() advances the spool kinematics (if driven) and decides whether the outermost wound point should be released during the current physics step. The procedure has two coupled effects: it updates the global angular phase θ_{off} of the helix, and it triggers discrete topology changes in the set of wound points. Its main tasks are:

- **Kinematic update of the spool**: if the simulation is in the deployment phase and the user has enabled cinematic driving, the function applies a constant angular speed $\omega =$ `spoolAngularSpeed` about the spool axis over the physics time step $\Delta t =$ `Time.fixedDeltaTime`. The accumulated angular offset is then advanced by:

$$\Delta\theta \;=\; \omega\,\Delta t, \qquad \theta_{\text{off}} \leftarrow \theta_{\text{off}} + \Delta\theta$$

- **Geometric release criterion**: if there is at least one point beyond the minimum which is still unwound, the function examines the outermost wound index, forms the corresponding arclength and evaluates the radial direction of the helix at that location:

$$\begin{aligned} i_{outermost} &= N_{wrap} - 1 \\ s_{outermost} &= L_{seg} \;\cdot\; i_{outermost} \\ \bar{n}_r(s_{outermost}, \theta_{off}) &= \text{HelixRadialDir}(s_{outermost}, \theta_{offset}) \end{aligned}$$

  Let $\hat{b} =$ `bottomDir` denote the precomputed unit vector pointing toward the gravitationally lowest point on the cylindrical surface. The code releases point $i_{outermost}$ when the radial direction lies within an angular window $\Delta\theta_{\text{win}}$ (in radians) around $\hat{b}$, equivalently when:

$$\bar{n}_r \cdot \hat{b} \;\geq\; \cos\Big(\Delta\theta_{\text{win}}\Big)$$

  Physically, this rule illustrates that the tether should detach near the bottom of the spool. Upon release, the point becomes dynamical: the flag `isLocked` is cleared, the previous position is set to the current one (to avoid artificial impulses in the ensuing Verlet step), the inverse mass is restored to the free-point value, and the wound count decrements.

In conclusion, if the wound count has reached the imposed minimum, the function invokes FinishDeployment( ) to close the deployment phase.

6.2.7 FinishDeployment( )

This terminal routine formalizes the end of the deployment phase. When the number of wound points falls to the minimum value, the code marks deployment as inactive and complete.

In the end, in order to physically stop the deployment, the spool is brought to rest by zeroing its angular velocity and putting it to sleep.

6.2.8 Awake( )

It initializes renderer and rigid-body references, validates rope length, constructs rope state, computes material and mass parameters, allocates solver buffers, draws the initial polyline, as in VerletRope. Additionally, **for the deployed case**, it initializes spool geometry (including axis, radial reference, helix pitch, turn length, and bottom direction), derives the number of initially wrapped points from excess length and marks them as locked to the spool, sets line positions using a helical placement for wrapped segments and a straight or extrapolated segment for the free part.

6.2.9 Start( )

It performs setup that depends on the runtime, after other scripts and physics have updated, similar to VerletRope. Additionally, **in the deployed case**, it recomputes spool geometry and the count of wrapped points if the rope length changes during runtime. It also initializes CSV logging by creating or overwriting the file and writing the header to record time, end-point position, and peak tension. Finally, it sets deployment state flags such as deploymentActive and deploymentComplete based on the wrapped points, which enables the timed release from the spool that follows.

6.2.10 OnValidate( )

It maintains editor-time consistency by clamping parameters and rebuilding rope/material/solver state for Scene preview (as in VerletRope). Additionally, with respect to the deployed case, recomputes spool geometry and wrapped-point state in the Editor (no Play Mode required); renders the helical wrap so the inspector reflects spool-driven initial conditions.

6.3 PID tension control during deployment phase

The controller reads the tension value from the XPBDTetherWithSpool simulation, filters it, computes the error with respect to the desired tension, and converts the control accelerations into tilt angles for the bench in order to apply the correction. The algorithm operates in the real system; only before converting the control accelerations into angles does it translate their value into the table's reference frame. The script also supervises spool deployment and can halt the spool once a target deployed length is reached.

The control diagram is fully analogous to that of a PID in the case of an already deployed tether:

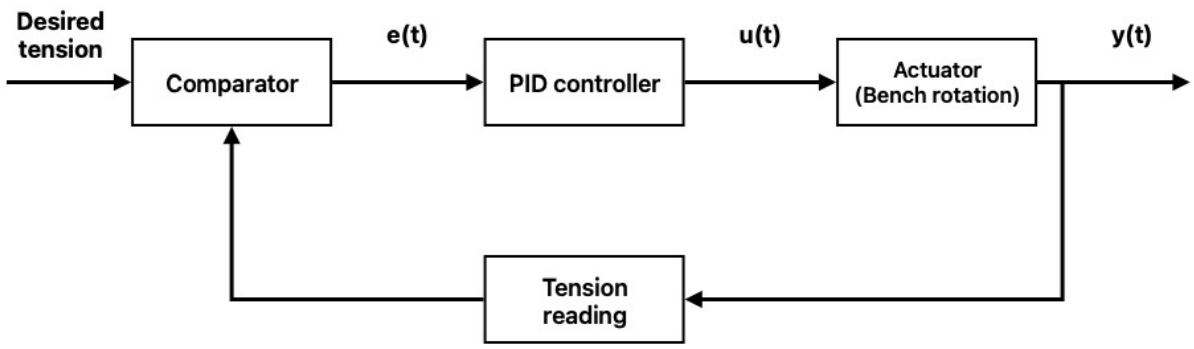


Figure 6.3: Deployment PID tension control diagram

The functions implemented are identical to those of the previously deployed tether case, with the only difference that, as soon as $L_{des} \geq L_{deployed}$, the deployment process is halted, while the PID tension control continues to operate.

6.3.1 Results of PID-Based tension control during tether deployment

The deployment phase represents a particularly delicate stage in the operation of tethered systems, since it involves both the dynamic release of the tether and the regulation of its mechanical tension. During this phase, the free length of the tether increases with time, and the system is subject to strong transient effects such as accelerations, oscillations, and potential whiplash phenomena. The control objective is therefore twofold: on one hand, ensuring that the tether is deployed in a smooth and stable manner without generating excessive tension peaks, and on the other, guaranteeing that the process halts precisely once the desired deployed length has been reached.

The try-and-error strategy for the PID tension gains led to the following values:

$$K_p = 1.9$$
$$K_i = 0.05$$
$$K_d = 1$$

In the digital twin, the test was carried out using the following values:

$$T_{des} = 0.05\ N$$
$$L_{deployed\,des} = 1\ m$$
$$L_{tether} = 1.16\ m$$
$$\bar{p}_{start} = (0,\ 0.9,\ 0)$$
$$f_{cutoff} = 3\ Hz$$
$$\beta_{max} = 2°$$
$$t_{actuators} = 2\ s$$
$$\omega_{spool} = 1\ rpm$$

The following results were obtained:

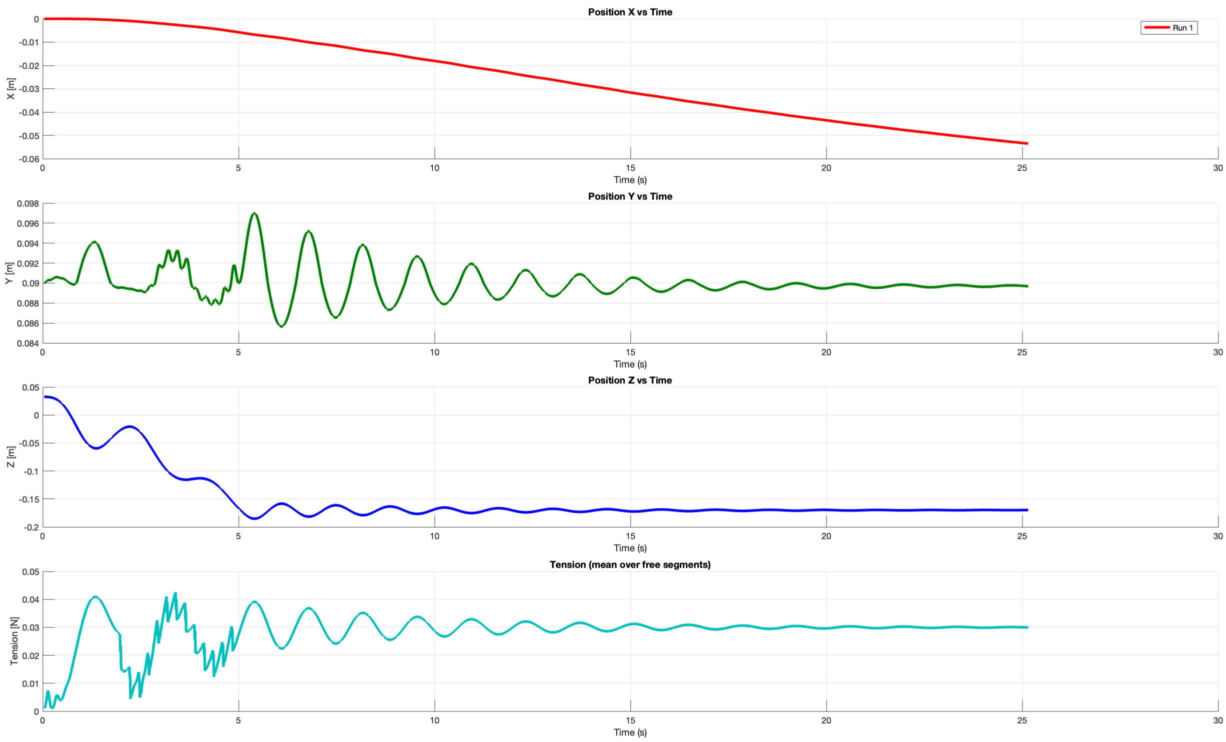


Figure 6.4: Deployment case - PID tension results

The analysis of the results reveals that the commanded tether length of 1 m is effectively achieved after approximately 5 seconds. This convergence can be identified by the disappearance of the discontinuities observed in the tension profile. During the deployment phase, the tension signal is characterized by a sequence of sharp transients, which manifest in the plots as sudden jumps.

These fluctuations arise from the discrete nature of the tether model: as each individual segment is progressively released from the spool, the effective free length of the tether undergoes an increase, leading to instantaneous drops in the transmitted tension. Such behavior is consistent with the physical expectation of stepwise deployment in a discretized representation of a flexible tether. Once the deployment process is completed, the tension response stabilizes, and the PID controller successfully drives the system towards the prescribed equilibrium condition, ensuring convergence to the desired steady-state tension.

PID tension control is crucial during the tether's deployment phase. When releasing the tether, maintaining a controlled and uniform tension is essential to prevent slack formation or uncontrolled whipping. These conditions can cause mechanical stress, oscillations, or even collisions between the connected bodies. By continuously monitoring the measured tether tension, the controller stabilizes the deployment rate and suppresses transient oscillations.

Chapter 7

Conclusions and future perspectives

Final considerations

This thesis has addressed the development and validation of a digital twin of an air-bearing platform for tethered satellite systems, with a specific focus on the two most critical operational phases: tether deployment and post-deployment control. The work was motivated both by the challenges identified in the literature review, as increasing the tether's survivability in LEO and the likelihood of a successful deployment.

The research began with an analysis of the low Earth orbit (LEO) environment. Particular attention was devoted to external perturbations such as atomic oxygen erosion, plasma interactions, radiative effects, and thermal fluctuations, all of which contribute to the degradation of tether materials and the complexity of long-term stability. This preliminary investigation provided a solid foundation for the subsequent Analytic Hierarchy Process (AHP) analysis, which was employed to systematically compare alternative materials, geometries, and deployment systems. Thanks to this type of analysis it was possible to evaluate both quantitative parameters and qualitative aspects. The resulting prioritization supported the identification of the most suitable tethered satellite system architecture for the two specified objectives.

On the modeling side, an AHP analysis was conducted to compare three different 3D simulations, leading to the selection of Unity™. Implementing the digital twin in the Unity™ environment turned out to be an effective and flexible solution. The platform allowed to combine 3D modeling, physics-based simulations, and control algorithm development in one environment. Specific challenges were faced during the numerical simulation of tether dynamics. The tether, unlike rigid bodies, needed a formulation that maintained its geometric constraints while ensuring stable and realistic oscillatory behavior. To tackle this, Verlet and Euler integrators, as well as Position-Based Dynamics (PBD) and its extended formulation (XPBD), were compared. The chosen combination of Verlet integration with the XPBD method showed better stability and accuracy in simulating tether dynamics, as indicated by the results of the numerical experiments.

A crucial aspect of this thesis has been the study of control strategies for both tether deployment and post-deployment operations. During the deployment phase, a PID controller was implemented to regulate tether tension, ensuring smooth release and avoiding dangerous transients such as slack tether or excessive whiplash effects. The post-deployment scenario, characterized by a constant free length of tether, was addressed through both the implementation of a PID controller and of a Linear Quadratic Regulator (LQR) for position control of the system. The simulation results confirmed the effectiveness of the implemented strategies, with the PID controller successfully maintaining tension close to the desired value, and the LQR e PID approaches providing stable and accurate control of the tethered floater relative motion.

The numerical results from the digital twin matched the expected physical behavior of tether systems. The simulated tether dynamics showed realistic responses regarding oscillations, vibrations, and tension changes. This shows that the chosen modeling framework can replicate the key physical phenomena seen in real systems. This outcome proves that the digital twin is a dependable virtual counterpart to the air-bearing platform. It can enhance experimental investigations and help validate control laws and deployment methods.

Overall, this work shows the benefits of combining ground-based experiments with digital twin environments. Air-bearing platforms offer a unique way to mimic planar microgravity conditions safely and affordably. The digital twin used in Unity increases this capability by providing flexibility, scalability, and constant access for simulation and analysis. Together, these two approaches create a strong tool for studying tethered satellite systems. They help reduce technological risks and development costs.

In conclusion, the research presented in this thesis has not only demonstrated the feasibility of constructing a digital twin for tethered satellite systems but has also shown its capability to replicate realistic behaviors under both deployment and post-deployment conditions. The results obtained confirm the validity of the digital twin approach, positioning it as a promising methodology to accelerate the design, analysis, and verification of tether-based space technologies.

Future perspectives

Several directions for future developments can be identified.

One avenue involves integrating detailed sensor models into the simulation environment. These models are essential in experimental setups. Including accurate models of sensors, such as noise, biases, and delays, would allow for testing under more realistic operating conditions. This would significantly improve the reliability of simulation results and reduce the gap between numerical predictions and experimental data.

Additionally, implementing control strategies offers a key opportunity. The current work showed the effectiveness of a PID controller for tether deployment, along with LQR and PID algorithms for post-deployment stabilization. Future studies could explore more advanced methods.

Besides deployment, a promising direction is implementing tether rewinding models. The analysis has focused on tether deployment and post-deployment dynamics, while retrieval has not been fully addressed. Modeling the rewinding process brings new challenges, like managing tether slack, controlling reeling torque, and accounting for extra friction and dynamic loads that differ from those experienced during deployment.

Finally, refining the dynamic models employed in the digital twin would further enhance the realism of the simulation. In particular, the deployment phase could benefit from incorporating frictional effects between the tether and the spool, as well as a more accurate representation of the mechanical properties of the winch system. Similarly, the current representation of the rotating deployment system and the floater is limited to simplified geometrical approximations. Developing more detailed models would improve the predictive accuracy of the simulation and enable a more reliable translation of numerical findings to physical experimental setups.

Bibliography

[1] M. Li Vigni. *Tethered Sitellite Systems: Missions Survey and Active Debris Removal Applications*. Bachelor Thesis. Turin, Italy, Sept. 2024 (cit. on p. 2).

[2] U. Bindra and Z.H. Zhu. «Ground based testing of space tether deployment using an air bearing inclinable turntable». In: *Int. J. Space Science and Engineering* 4.1 (2016) (cit. on p. 4).

[3] A. Francesconi, C. Giacomuzzo, F. Branz, and E.C. Lorenzini. «Sur-vivability to hypervelocity impacts of electrodynamic tape tethers for deorbiting spacecraft in LEO». In: *6th European conference on space debris*. Darmstadt, Germany, 2013 (cit. on p. 4).

[4] S.B. Khan and J.R. Sanmartin. «Survival probability of round and tape tethers against debris impact». In: *Journal of Spacecraft and Rockets* 50.3 (2013), pp. 603–608 (cit. on p. 4).

[5] A. Brunello, L. Olivieri, G. Sarego, A. Valmorbida, E. Lungavia, and E.C. Lorenzini. «Space tethers: parameters reconstructions and tests». In: *IEEE 8th International Workshop on Metrology for AeroSpace (MetroAeroSpace)*. 2021 (cit. on p. 4).

[6] C. Menon, M. Kruijff, and A. Vavouliotis. «Design and testing of a space mechanism for tether deployment». In: *Journal of Spacecraft and Rockets* 44.4 (2007), pp. 927–939 (cit. on pp. 5, 18).

[7] L. Johnson, B. Gilchrist, E. Lorenzini, and N. Stone. «Propulsive small expandable deployer system (ProSEDS) experiment: mission overview and status». In: *39th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*. Huntsville, Alabama, 2003 (cit. on p. 5).

[8] Wikipedia. *Young Engineers' Satellite 2*. Accessed: 2025-05-8. 2025. URL: https://en.wikipedia.org/wiki/Young_Engineers%27_Satellite_2 (cit. on p. 5).

[9] Y. Yang, K. Yang, J. Zhang, H. Cai, C. Zhou, and L. Li. «A Novel Design and Optimization Method for an Electrodynamic Tether Deployment Mechanism». In: *Space: Science Technology* 4 (Apr. 2024) (cit. on p. 5).

[10] H.Z. Zhu, J. Kang, and U. Bindra. «Validation of CubeSat tether deployment system by ground and parabolic flight testing». In: *Acta Astronautica* 185 (2021), pp. 299–307 (cit. on p. 5).

[11] M. Shan and L. Shi. «Comparison of Tethered Post-Capture System Models for Space Debris Removal». In: *Aerospace* 9.33 (2022) (cit. on p. 6).

[12] M. Becker, E. Stoll, K. Soggeberg, and I. Retat. «Approaches and models for flexible tether conncetions in active debris removal missions». In: *7th European Conference on Space Debris*. Darmstadt, Germany, 2017 (cit. on pp. 6, 7).

[13] M. Kruijff, E.J. Van der Heide, and M. Stelzer. «Applicability of tether deployment simulation and tests nased on YES2 flight data». In: *AIAA Modeling and Simulation Technologies Conference and Exhibit*. Honolulu, Hawaii, 2008 (cit. on p. 7).

[14] Wikipedia. *MSC Adams*. Accessed: 2025-05-2. 2025. URL: https://en.wikipedia.org/wiki/MSC_Adams (cit. on p. 7).

[15] MathWorks. *Simscape Multibody*. Accessed: 2025-05-2. 2025. URL: https://en.wikipedia.org/wiki/MSC_Adams (cit. on p. 7).

[16] M. Kruijff, E.J. Van der Heide, and M. Stelzer. «Novel dynamic model for an object-oriented space tether simulator"». In: *ISSFD-25th*. Munich, Germany, 2015 (cit. on p. 7).

[17] GitHub. *Welcome to Basilisk: an astrodynamics simulation framework*. Accessed: 2025-05-2. 2025. URL: https://avslab.github.io/basilisk/ (cit. on p. 7).

[18] NASA. *Simulation Tools*. Accessed: 2025-05-2. 2025. URL: https://www.nasa.gov/general/simulation-tools/ (cit. on p. 7).

[19] NVIDIA. *Physics simulation fundamentals*. Accessed: 2025-09-o6. URL: https://docs.isaacsim.omniverse.nvidia.com/4.2.0/simulation_fundamentals.html (cit. on p. 7).

[20] M. Singh, J. Kapukotuwa, E.L.S. Gouveia, E. Fuenmayor, Y. Qiao, N. Murray, and D. Devine. «Comparative Study of Digital Twin Developed in Unity and Gazebo». In: *Electronics* 14.276 (2025) (cit. on pp. 7, 8, 28).

[21] C. Bua, L. Borgianni, D. Adami, and S. Giordano. «Reinforcement Learning-Driven Digital Twin for Zero-Delay Communication in Smart Greenhouse Robotics». In: *Agriculture* 15.1290 (2025) (cit. on pp. 8, 31).

[22] G. Governale, J. Rimani, N. Viola, and V. Fernandez Villace. «A trade-off methodology for micro-launchers». In: *Aerospace Systems* 4 (2021), pp. 209–226. DOI: 10.1007/s42401-021-00095-w. URL: https://doi.org/10.1007/s42401-021-00095-w (cit. on p. 10).

[23] Songhan. *Honeywell Spectra® 900 Fiber*. Accessed: 2025-05-16. URL: https://www.lookpolymers.com/pdf/Honeywell-Spectra-900-Fiber.pdf (cit. on p. 12).

[24] Mehler. *Investigations by Mehler on the PBO-Fiber Zylon® from Toyobo*. Accessed: 2025-05-16. URL: https://media.cdn.lexipol.com/_misc/BodyArmor/MehlerZylon_DSM.pdf (cit. on p. 12).

[25] DuPont. *Kevlar® Aramid Fiber Technical Guide*. Accessed: 2025-05-16. URL: https://www.dupont.com/content/dam/dupont/amer/us/en/safety/public/documents/en/Kevlar_Technical_Guide_0319.pdf (cit. on p. 12).

[26] Honeywell. *Honeywell Spectra® Fiber Capability Guide*. Accessed: 2025-05-16. URL: https://prod-edam.honeywell.com/content/dam/honeywell-edam/pmt/oneam/en-us/high-performance-fibers/documents/SpectraFiber-CapabilityGuide-Brochure.pdf (cit. on p. 12).

[27] Toyobo. *Zylon® (PBO Fiber) Technical Information 2005*. Accessed: 2025-05-16. URL: https://www.toyobo-global.com/seihin/kc/pbo/zylon-p/bussei-p/technical.pdf (cit. on p. 12).

[28] NASA. *Atomic Oxygen Erosion Yield Prediction for Spacecraft Polymers in Low Earth Orbit*. Accessed: 2025-05-16. URL: https://ntrs.nasa.gov/api/citations/20090034484/downloads/20090034484.pdf (cit. on p. 12).

[29] K.A. Gittemeier, C.W. Hawk, M.M. Finckenor, and E. Watts. «Low Earth Orbit Environmental Effects on Space Tether Materials». In: *American Institute of Aeronautics and Astronautics* () (cit. on p. 14).

[30] K.A. Gittemeier, C.W. Hawk, M.M. Finckenor, and E. Watts. «Space Environmental Effects on Coated Tether Materials». In: *American Institute of Aeronautics and Astronautics* () (cit. on p. 14).

[31] C. Pardini, T. Hanada, P.H. Krisko, L. Anselmo, and H. Hirayama. «Are de-orbiting missions possible using electrodynamic tethers? Task review from the space debris perspective.» In: *Acta Astronautica* 60 (2007), pp. 916–929 (cit. on p. 14).

[32] Y. Uwamino, M. Fujiwara, H. Tomizaki, K. Ohtani, and K. Makihara. «Damage of Twisted Tape Tethers on Debris Collision». In: *Internation Journal of Impact Engineering* 137.103440 (2020) (cit. on p. 15).

[33] Wikipedia. *Space tether missions*. Accessed: 2025-05-23. URL: `https://en.wikipedia.org/wiki/Space_tether_missions` (cit. on p. 15).

[34] J.L. Van Noord, B. West, and B. Gilchrist. «Electrodynamic Tape Tether Performance with Varying Tether Widths at Low Earth Altitudes». In: *American Institute of Aeronautics and Astronautics* () (cit. on p. 15).

[35] P. Toivanen, P. Janhunen, J. Kivekäs, and M. Mäkelä. «Robust Flight Tether for In-Orbit Demonstrations of Coulomb drag Propulsion». In: *Aerospace* 11.62 (2024) (cit. on p. 15).

[36] G. Sarego et al. «Deployment requirements for deorbiting electrodynamic tether technology». In: *CEAS Space Journal* 13 (2021) (cit. on pp. 18, 19).

[37] Yi Yang, Keying Yang, Jingrui Zhang, Han Cai, Chunyang Zhou, and Lincheng Li. «A Novel Design and Optimization Method for an Electrodynamic Tether Deployment Mechanism». In: *Space: Science & Technology* 4 (2024). Published April 25, 2024, Article 0147. DOI: `10.34133/space.0147`. URL: `https://doi.org/10.34133/space.0147` (cit. on p. 19).

[38] National Space Society. *Tethers*. Accessed: 2025-05-20. URL: `https://www.nss.org/settlement/nasa/spaceresvol2/tethers.html` (cit. on p. 21).

[39] European Space Agency. *Snap-proof space tether*. Accessed: 2025-05-20. URL: `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Snap-proof_space_tether` (cit. on p. 21).

[40] D.D. Tomlin, G.C. Faile, K.B. Hayashida, C.L. Frost, C.Y. Wagner, M.L. Mitchell, J.A. Vaughn, and M.J. Galuska. *Space tethers: Design Criteria*. Technical Memorandum TM-1997-108537. MSFC, Alabama: National Aeronautics and Space Administration (NASA), July 1997. URL: `https://ntrs.nasa.gov/api/citations/19970027081/downloads/19970027081.pdf` (cit. on pp. 21, 22).

[41] J.A. Carroll. *Guidebook for analysis of tether applications*. Ed. by National Aeronautics and Space Administration (NASA). San Diego, CA, 1985. URL: `https://ntrs.nasa.gov/api/citations/19870001511/downloads/19870001511.pdf` (cit. on p. 21).

[42] M. Kruijff. «Tethers in Space: A Propellantless Propulsion In-Orbit Demonstration». PhD thesis. Delft, The Netherlands: Technische Universiteit Delft, May 2011. URL: `http://resolver.tudelft.nl/uuid:01b6e4d4-8d9e-4f51-b0e6-993d0f5e1f0e` (cit. on p. 22).

[43] R. Walsh and C.A. Swenson. «Mechanical Properties of Zylon/Epoxy Composite at 295K and 77 K». In: *IEEE Transactions on Applied Superconductivity* 16.2 (2006), pp. 1761–1764 (cit. on p. 24).

[44] Teijin Frontier (U.S.A),INC. *PBO Fiber (ZYLON®)*. Accessed: 2025-05-30. URL: `https://www.teijin-frontier-usa.com/zylon/` (cit. on p. 24).

[45] Blender. *Blender 3.4 Manual – Getting Started*. Accessed: 2025-09-08. URL: `https://docs.blender.org/manual/en/3.4/getting_started/about/introduction.html` (cit. on p. 27).

[46] Erwin Coumans. *Bullet 2.80 Physics SDK Manual.* Accessed: 2025-09-08. URL: `https://www.cs.kent.edu/~ruttan/GameEngines/lectures/Bullet_User_Manual` (cit. on p. 27).

[47] Wikipedia. *Python (programming language).* Accessed: 2025-09-06. URL: `https://en.wikipedia.org/wiki/Python_%28programming_language%29` (cit. on p. 28).

[48] J. Collins, S. Chand, A. Vanderkop, and G. Howard. «A Review of Physics Simulators for Robotic Applications». In: *IEEE Access* PP (Mar. 2021), pp. 1–1. DOI: `10.1109/ACCESS.2021.3068769` (cit. on p. 28).

[49] Unity Technologies. *Unity real-time development platform.* Accessed: 2025-09-06. URL: `https://unity.com/` (cit. on p. 28).

[50] NVIDIA. *NVIDIA IsaacSim.* Accessed: 2025-09-06. URL: `https://developer.nvidia.com/isaac/sim` (cit. on p. 30).

[51] NVIDIA. *Installation Requirements — NVIDIA Isaac Sim 4.5.0 Documentation.* Accessed: 2025-09-06. URL: `https://docs.isaacsim.omniverse.nvidia.com/4.5.0/installation/requirements.html` (cit. on p. 30).

[52] NVIDIA developer forum. *What Is the Correct Way to Build a Simulated Rope?* Accessed: 2025-09-06. URL: `https://forums.developer.nvidia.com/t/what-is-the-correct-way-to-build-a-simulated-rope/244881` (cit. on p. 30).

[53] G. Governale, A. Pastore, M. Clavolini, M. Li Vigni, C. Bellinazzi, C. L. Matonti, S. Aliberti, R. Apa, and M. Romano. «Hardware-in-the-Loop Testing of Spacecraft Relative Dynamics and Tethered Satellite System on a Tip-Tilt Flat-Table Facility». In: *Aerospace* 12 (2025), p. 884. DOI: `10.3390/aerospace12100884`. URL: `https://doi.org/10.3390/aerospace12100884` (cit. on pp. 33, 34, 42).

[54] B.R. Fernandez, L. Herrera, J. Hudson, and M. Romano. «Development of a tip-tilt air-bearing testbed for physically emulating proximity-flight orbital mechanics». In: *Advances in space research* 71 (2025), pp. 4332–4339 (cit. on p. 33).

[55] A.D. Ogundele, B.R. Fernandez, J. Virgili-Llop, and M. Romano. «A tip-tilt hardware-in-the-loop air-bearing test bed with physical emulation of the relative orbital dynamics». In: *Proceedings of the 29th AAS/AIAA Space Flight Mechanics Meeting* 168 (2019), p. 3781 (cit. on p. 33).

[56] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms of Ordinary Differential Equations.* 2nd. Berlin: Springer, 2006 (cit. on p. 35).

[57] T. Jakobsen. «Advanced Character Physics». In: *Proceedings of the Game Developers Conference (GDC).* San Jose, CA, 2001 (cit. on p. 35).

[58] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* 2nd. Oxford: Oxford University Press, 2017 (cit. on p. 35).

[59] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. «Position Based Dynamics». In: *Journal of Visual Communication and Image Representation* 18.2 (2007), pp. 109–118 (cit. on p. 35).

[60] D. Baraff and A. Witkin. «Large Steps in Cloth Simulation». In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98).* New York: ACM, 1998, pp. 43–54 (cit. on p. 35).

[61] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim. «XPBD: Position-Based Simulation of Compliant Constrained Dynamics». In: *Proceedings of the 9th International Conference on Motion in Games (MIG '16).* New York: ACM, 2016, pp. 49–54 (cit. on pp. 35, 36).

[62] K. Seweryn, R. Rybus, J. Oleś, and K. Tarenko. *Validation Methodology of the Rendezvous and Grasping Manoeuvre on the Planar Air-Bearing Microgravity Simulator*. https://indico.esa.int/event/181/contributions/1421/attachments/1313/1538/Seweryn_etal_CS2017_v4.pdf. Accessed: 2025-08-31. 2017 (cit. on p. 41).

Appendix A

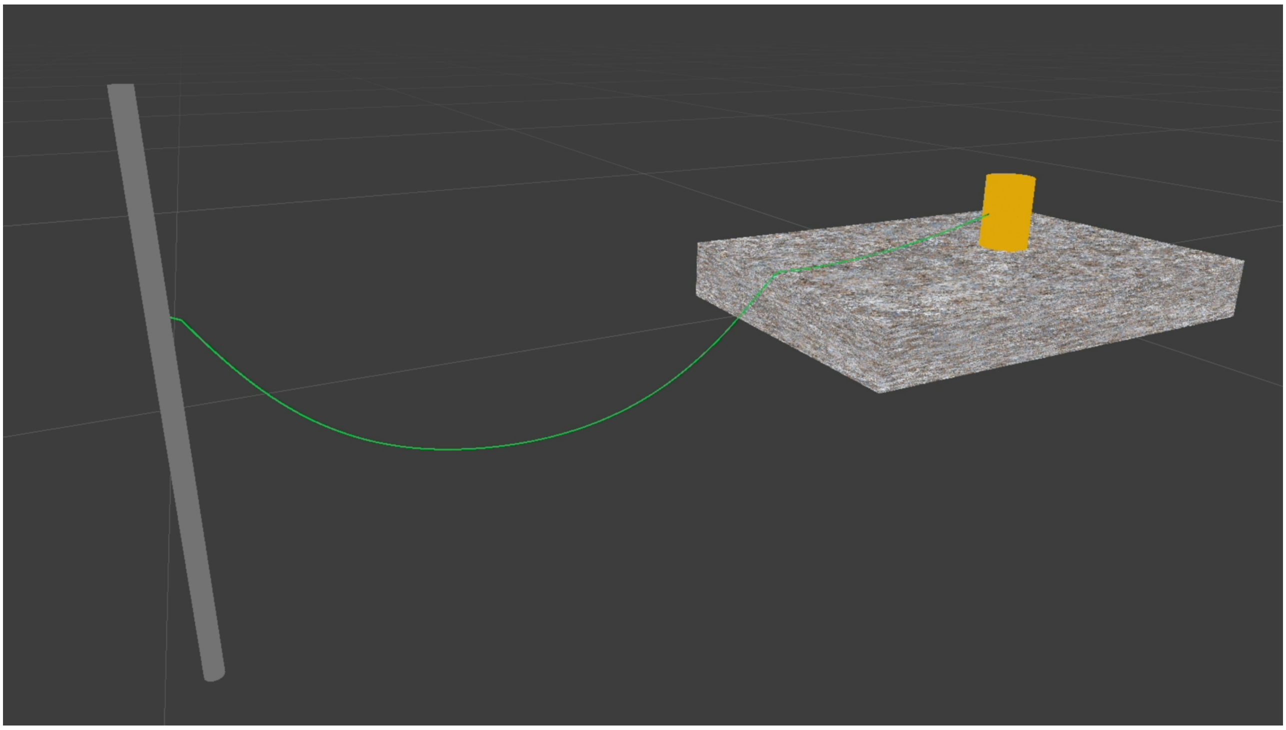
Results video

The videos of the results of the various tests performed are freely available at the following link:
`https://mega.nz/folder/FQVTQQiK#PlZRW0Ee2BZV3il40txn2g`

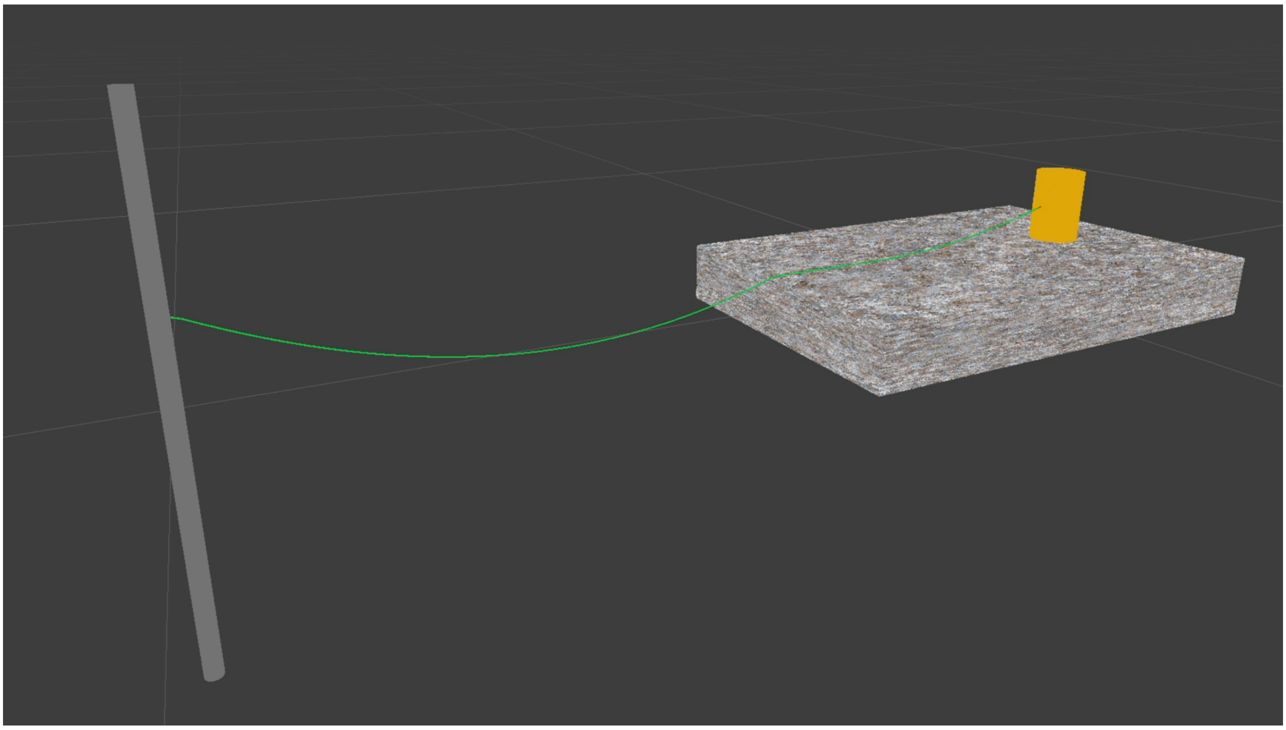
or using the QR code:

Figure A.1: Results QR code

PID tension control relevant frames - deployed case

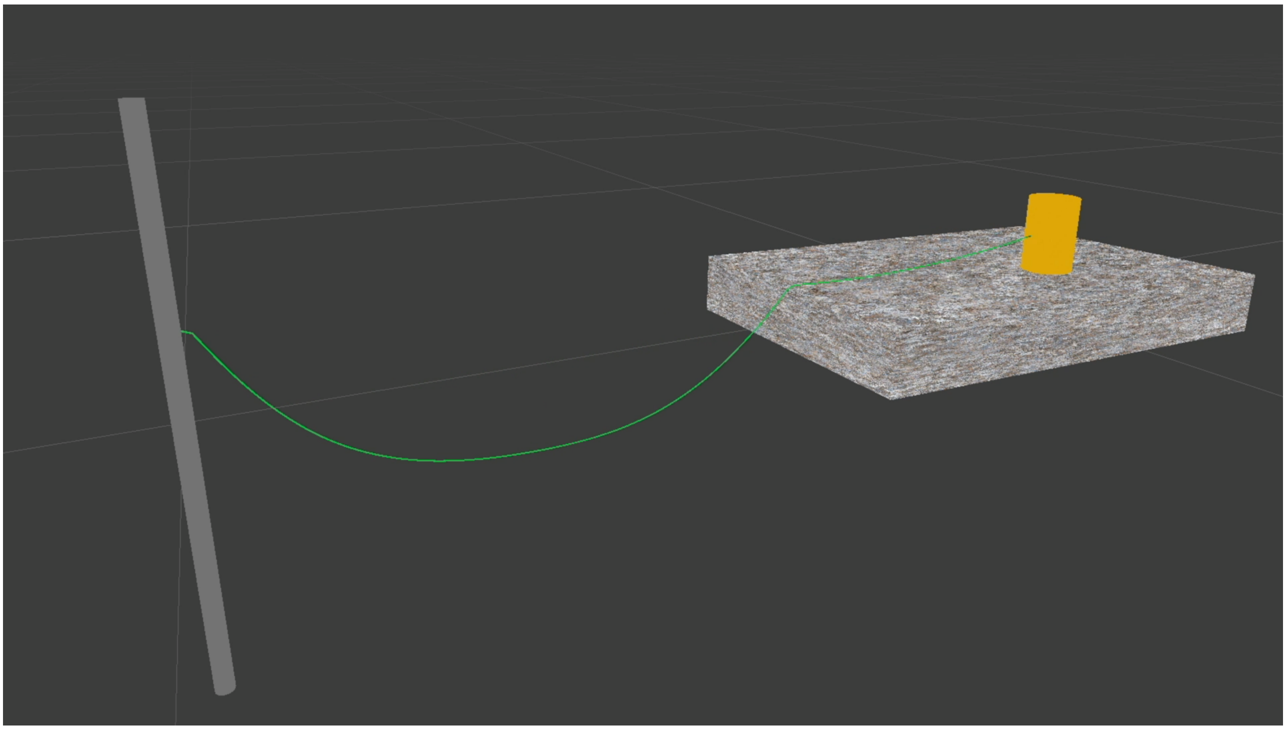


(a) PID tension - Initial bench inclination commanded by the controller

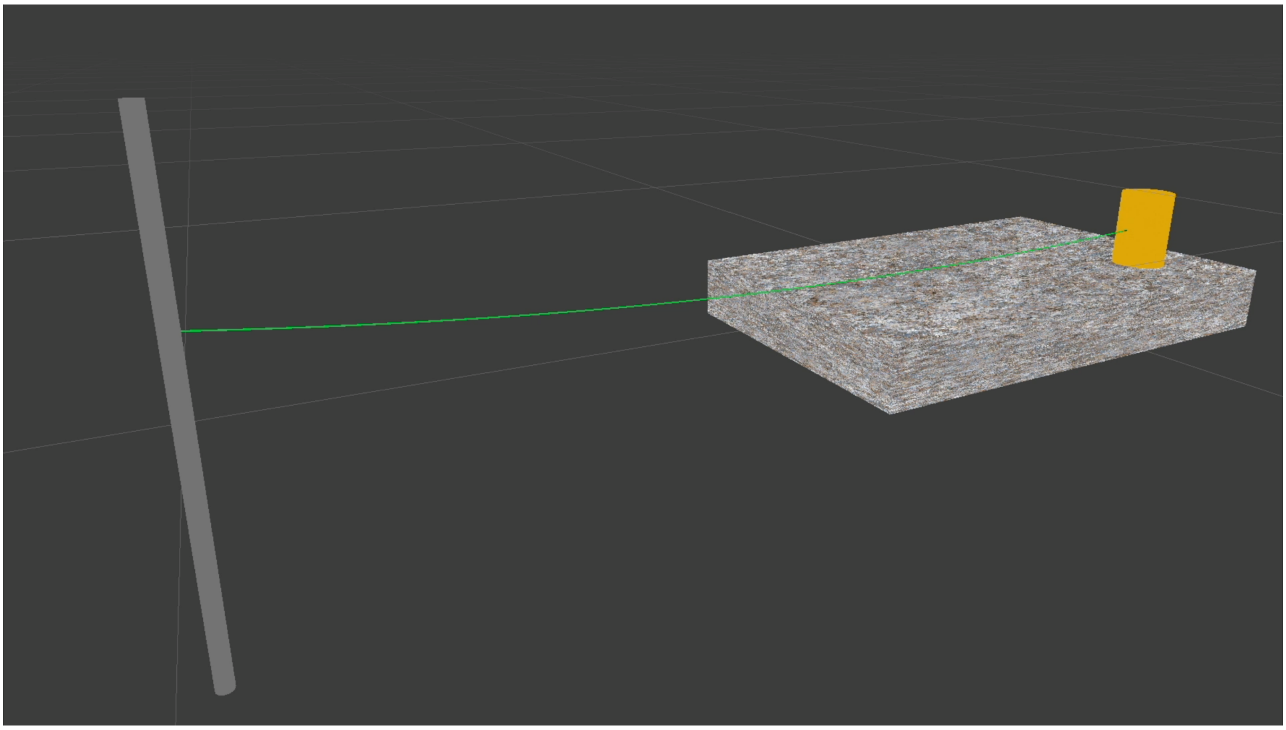


(b) PID tension - Achievement of the desired tension

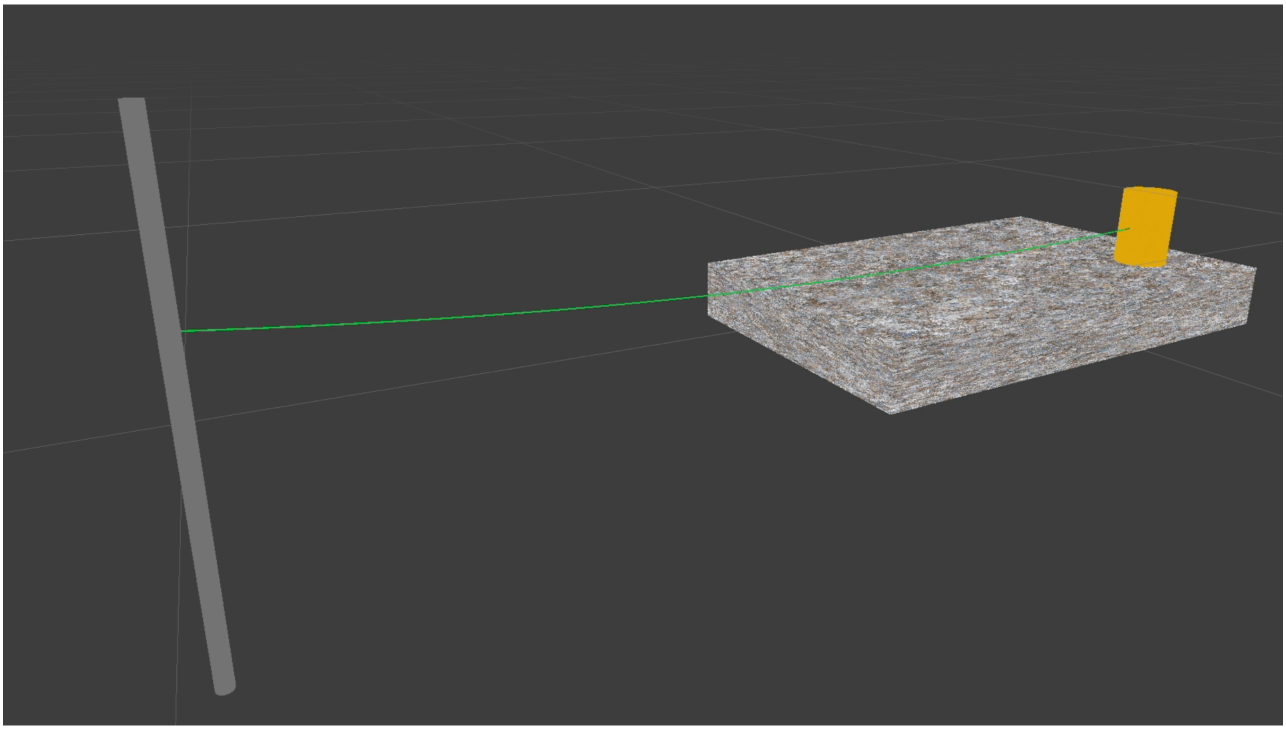
PID position control relevant frames - deployed case



(a) PID position - Initial bench inclination commanded by the controller

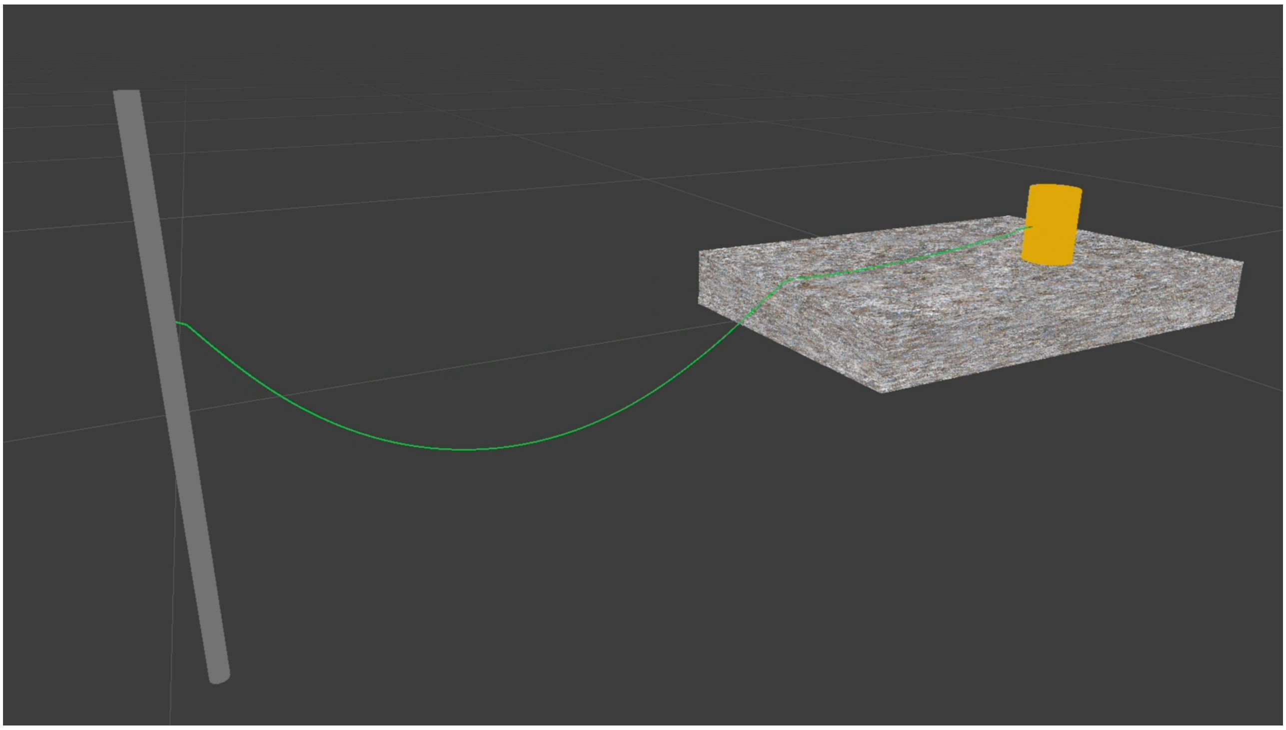


(b) PID position - Bench tilt around the z-axis

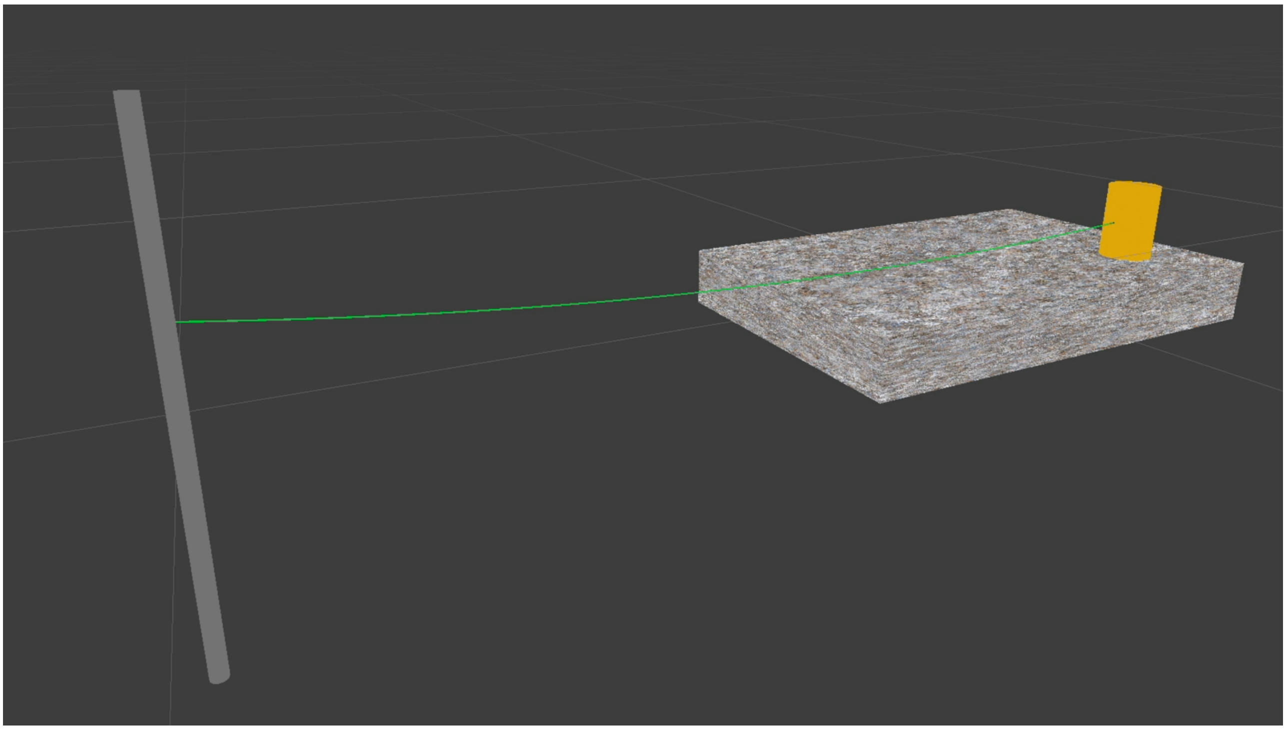


(c) PID position - Achievement of the desired position

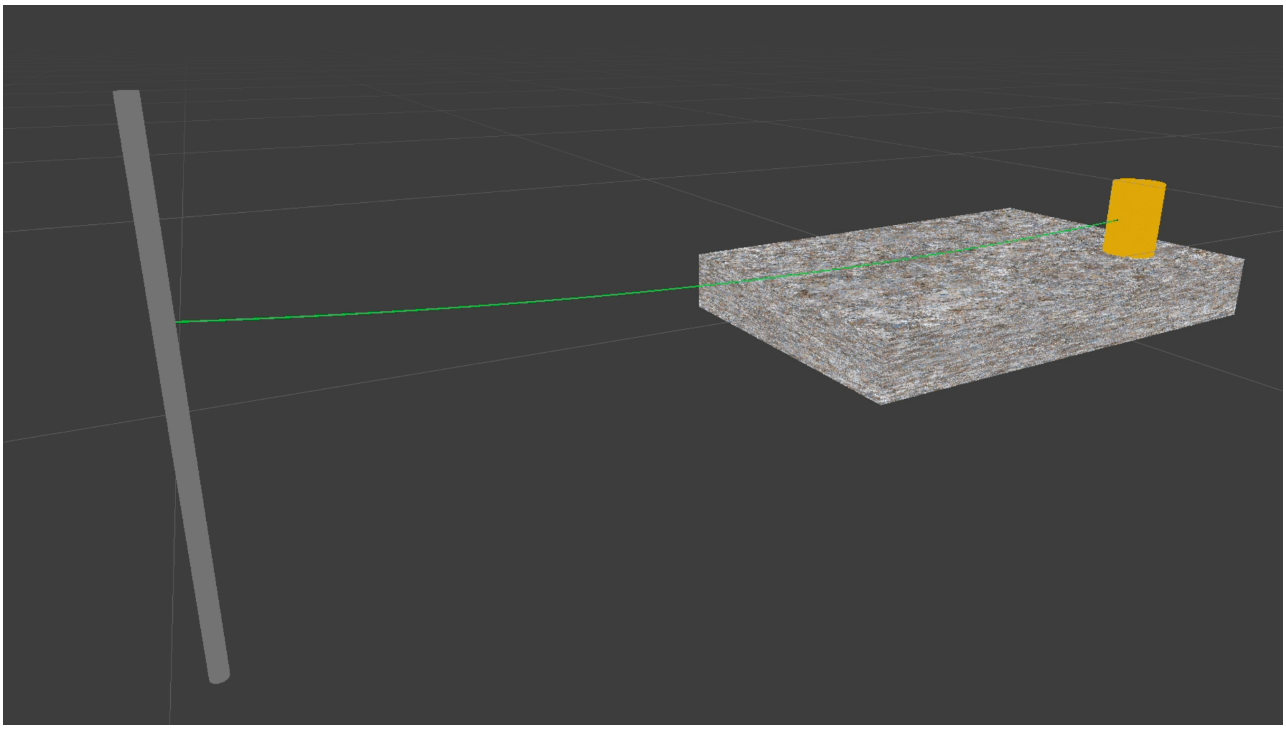
LQR position control relevant frames - deployed case



(a) LQR - Initial bench inclination commanded by the controller

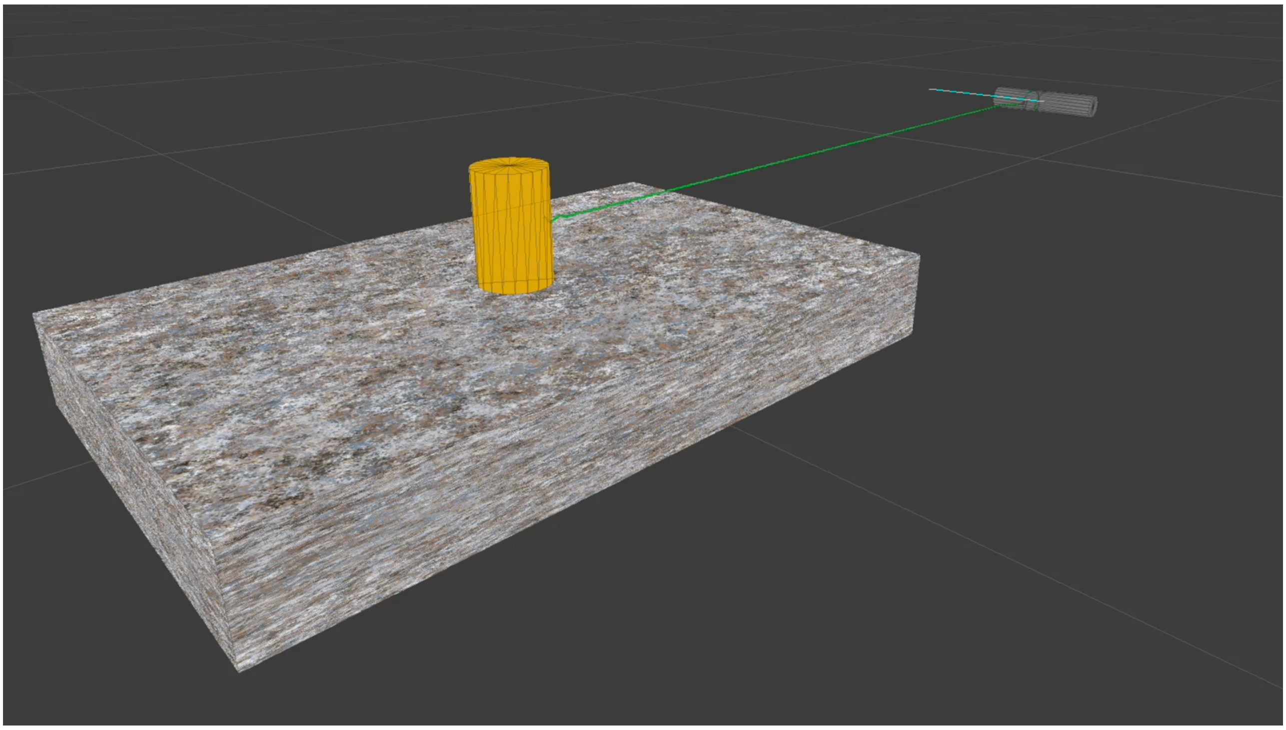


(b) LQR - Fine adjustment of the floater position (small bench tilt)

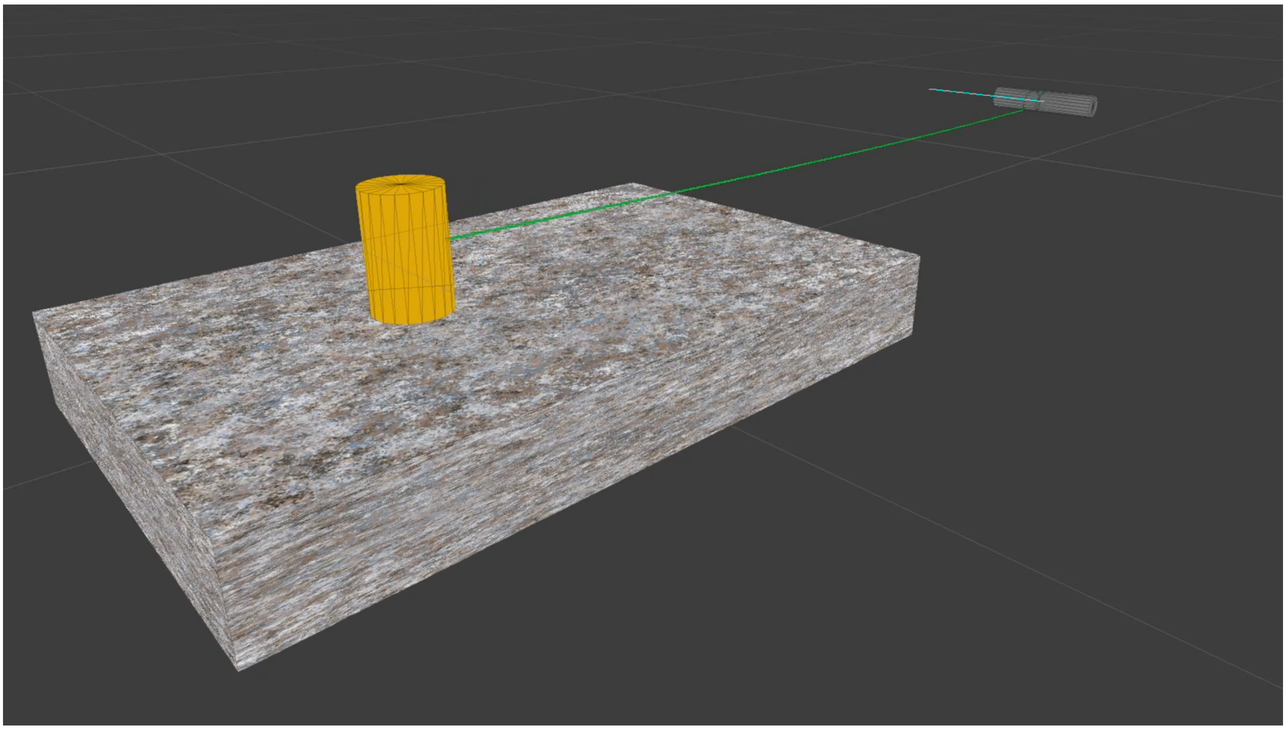


(c) LQR - Achievement of the desired position

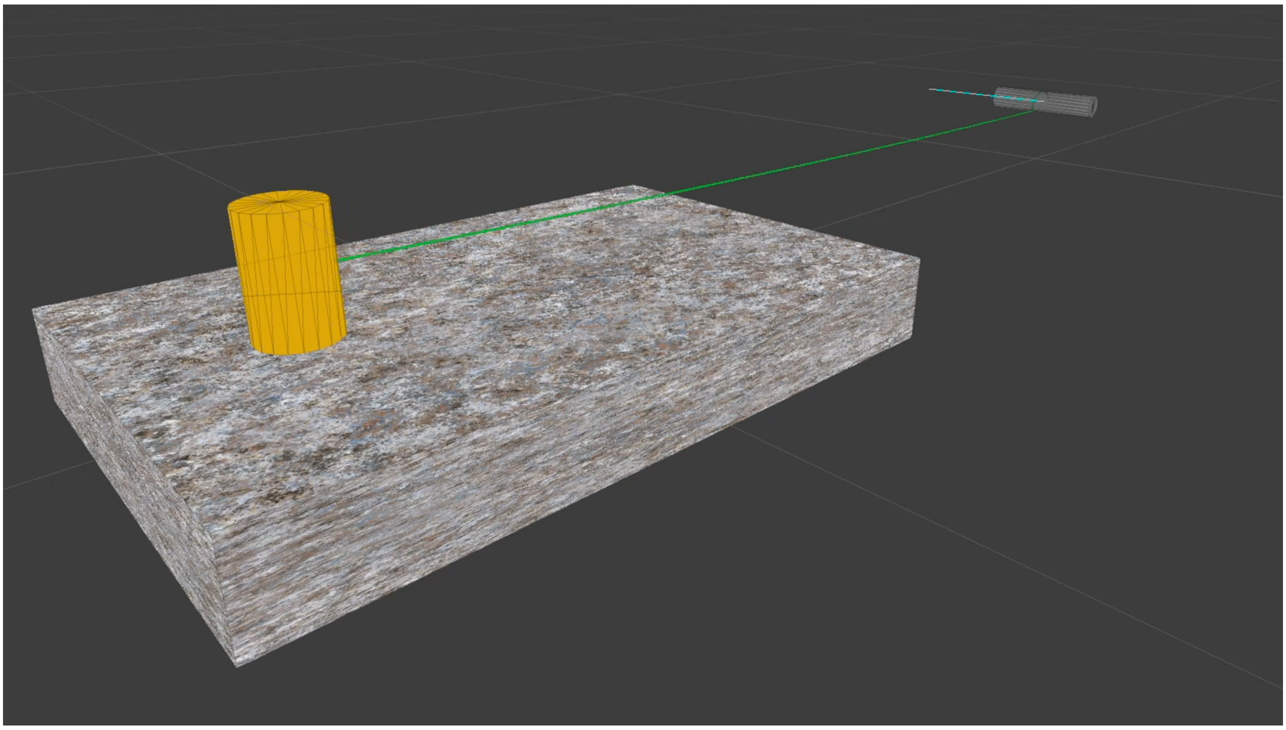
PID tension control relevant frames - deployment case



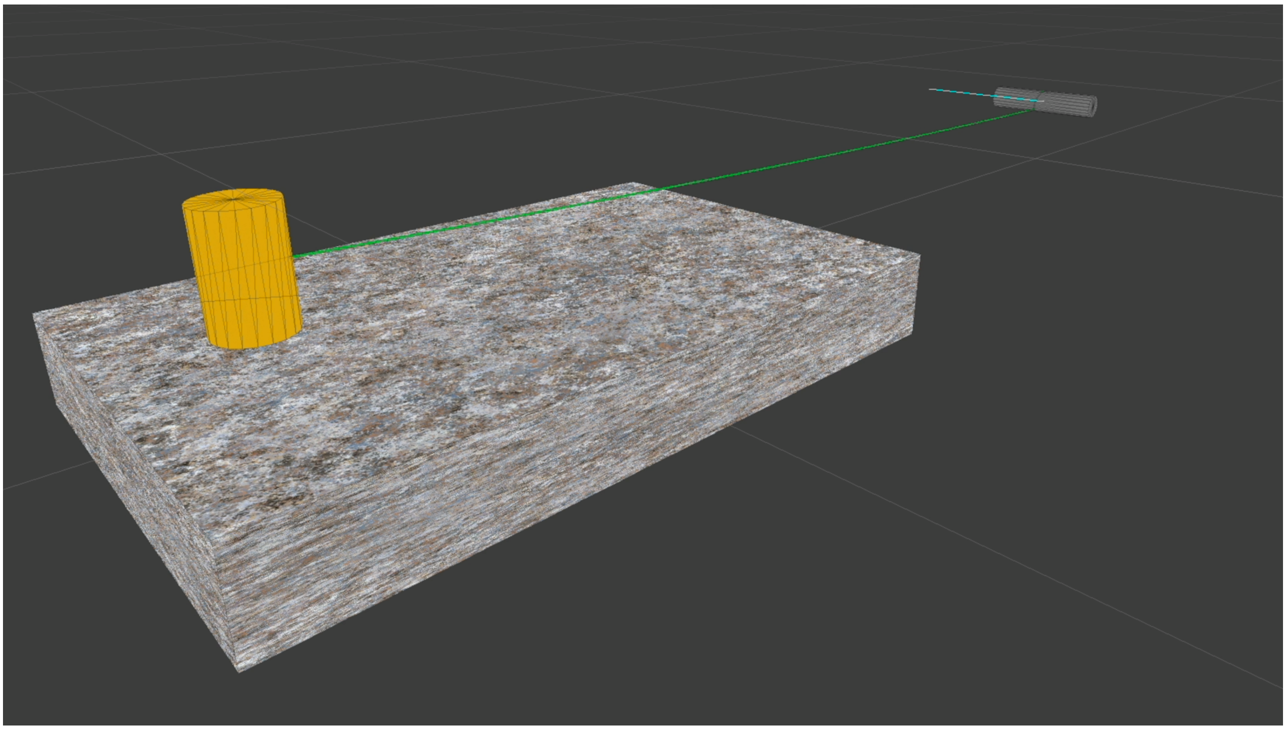
(a) Deployment - Initial configuration



(b) Deployment - Tension drop caused by the release of new tether segments



(c) Deployment - Achievement of the desired tether length



(d) Deployment - Achievement of the desired tension

Appendix B

VerletRope.cs

```
/*
 * Project    : A digital twin of an air-bearing platform for tethered
    satellite systems:
                 from tether deployment to post-deployment control
 * File       : VerletRope.cs
 * Author     : Edoardo De Blasi
 * Supervisors: Prof. Paolo Maggiore, Dr. Giuseppe Governale, Prof.
   Stephanie Lizy-Destrez
 * Date       : September 2025
 * Notes      : Developed in Unity using C# scripting.
 * License    : This code is intended for academic and research
   purposes only.
 */
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[DefaultExecutionOrder(10000)]
[RequireComponent(typeof(LineRenderer))]
public class VerletRope : MonoBehaviour
{
    // -------- Anchors --------
    [Header("Anchors")]
    public Transform startPointTransform;
    public Transform endPointTransform;

    // === Back-reaction & COM ===
    [Header("Back-reaction & Floater COM")]
    [SerializeField] private bool applyBackReaction = true;
    [SerializeField] private bool autoFindRigidbodies = true;
    public Rigidbody startRB;  // pole rigidbody
    public Rigidbody endRB;  // floater rigidbody
    [SerializeField] private bool setEndRbCenterOfMass = true;
    [SerializeField] private Vector3 endRbCenterOfMassLocalOffset =
   new Vector3(0f, -0.2f, 0f); // lowers the CoM
    [SerializeField] private bool setStartRbCenterOfMass = false;
    [SerializeField] private Vector3 startRbCenterOfMassLocalOffset =
    Vector3.zero;

```

```
    // -------- Tether --------
    [Header("Tether Settings")]
    [SerializeField, Min(2)] private int segmentCount = 35;

    [Tooltip("Tether length")]
    [SerializeField, Min(1e-4f)] private float ropeLengthMeters = 1f;

    [Tooltip("If false, it assigns the tether a length equal to the distance between the anchors")]
    [SerializeField] private bool useInspectorRopeLength = true;

    [Tooltip("If true and the anchors are farther apart than rop...djust ropeLength to the distance to avoid an impossible state")]
    [SerializeField] private bool clampRopeLengthToAnchors = true;

    [SerializeField] private bool snapStraightAtStart = true;
    [SerializeField] private float gravity = -9.81f;            // m/s^2
    [SerializeField, Range(0.90f, 1f)] private float damping = 0.997f;

    // -------- Material --------
    [Header("Material (Hooke)")]
    [SerializeField] private float youngModulus = 8.0e9f;   // Pa
    [SerializeField] private float ropeDiameter = 0.00016f; // m
    [SerializeField] private float density = 1540f;         // kg/m^3

    [Header("XPBD")]
    [SerializeField] private float compliance = 1e-10f;     // inverse stiffness (m/N)
    [SerializeField, Min(1)] private int xpbdIterations = 6;
    [SerializeField] private float constraintTolerance = 5e-5f; // m

    [Header("Integration")]
    [SerializeField, Min(1)] private int substeps = 8;
    [SerializeField] private float timeScale = 1f;
    [SerializeField] private float fixedDeltaTime = 0.02f;

    [Header("Collisions")]
    [SerializeField] private LayerMask collisionMask;
    [SerializeField, Range(1, 8)] private int samplesPerSegment = 3;
    [SerializeField] private float collisionRadius = 0.0015f;

    [Header("Rendering")]
    [SerializeField] private Color ropeColor = Color.white;
    [SerializeField] private float ropeWidth = 0.002f;

    [Header("Anchors")]
    [SerializeField] private Transform startPointTransform;
    [SerializeField] private Transform endPointTransform;

    [Header("Mass / Inertia")]
    [SerializeField] private Rigidbody floaterRb; // floater rigidbody
    [SerializeField] private Rigidbody spoolRb; // spool rigidbody
```

```

    [Header("Options")]
    [SerializeField] private bool autoFindRigidbodies = true;

    [Header("Rope Geometry")]
    [SerializeField, Min(2)] private int segmentCount = 60;
    [SerializeField] private float ropeLengthMeters = 2.0f;

    [Header("Anchoring / Locking")]
    [SerializeField] private bool lockFirstPoint = true;
    [SerializeField] private bool lockLastPoint = true;

    [Header("XPBD Tuning")]
    [SerializeField, Range(0.9f, 1f)] private float lambdaDecay =
0.98f; // lambda decay each frame
    [SerializeField] private bool scaleComplianceWithSubsteps = true;
     // c' = c / S^2

    // -------- Sanity check --------
    [Header("Sanity Check (log)")]
    // --- Sanity thresholds (new) ---
    [SerializeField] private float arcTolAbs = 1e-3f;       // 1 mm
    [SerializeField] private float arcTolRel = 2e-3f;       // 0.2% of
  L
    [SerializeField] private float sagVsSlackFactor = 3.5f;
    [SerializeField] private float warnConstraintFactor = 20f;
    [SerializeField] private bool enableSanityCheck = true;
    [SerializeField, Min(1)] private int logEveryNFrames = 30;

    [Tooltip("Maximum allowed sag (fraction of the total length).")]
    [SerializeField, Range(0.01f, 0.75f)] private float
maxSagFraction = 0.35f;

    [Tooltip("If true, logs a warning when the average constraint
error stays high for multiple substeps in a row.")]
    [SerializeField] private bool warnOnHighConstraintError = true;

    [Header("Output")]
    [HideInInspector] public float Tension { get; private set; }
    [HideInInspector] public float[] SegmentTensions { get; private
set; }

    // -------- Internals --------
    private LineRenderer lineRenderer;

    private struct RopePoint
    {
        public Vector3 currentPosition;
        public Vector3 previousPosition;
        public bool locked;
        public bool wound; // true if the point is wound on the spool
        public Vector3 force;
    }

    private RopePoint[] ropePoints;
```

```
    private float segmentRestLength;
    private float linearMass;  // kg/m
    private float EA;  // N (Young's modulus * area)
    private float effectiveCompliance;
    private float invMassPerPoint;

    private float[] lambdas;         // XPBD Lagrange multipliers per
segment
    private float[] segmentLengths;  // current segment lengths

    private int frameCounter;


    private IEnumerator Start()
    // It initializes rope objects and runs any startup coroutines.
    {
        if (autoFindRigidbodies)
        {
            if (!floaterRb && startPointTransform) floaterRb =
startPointTransform.GetComponentInParent<Rigidbody>();
            if (!spoolRb && endPointTransform) spoolRb =
endPointTransform.GetComponentInParent<Rigidbody>();
        }

        lineRenderer = GetComponent<LineRenderer>();
        if (!lineRenderer) lineRenderer = gameObject.AddComponent<
LineRenderer>();
        lineRenderer.positionCount = segmentCount + 1;
        lineRenderer.startWidth = ropeWidth;
        lineRenderer.endWidth = ropeWidth;
        lineRenderer.material = new Material(Shader.Find("Sprites/
Default"));
        lineRenderer.startColor = ropeColor;
        lineRenderer.endColor = ropeColor;
        lineRenderer.useWorldSpace = true;

        if (autoFindRigidbodies && !floaterRb && !spoolRb)
        {
            // No rigidbodies found automatically - this is fine if
rope endpoints are static.
        }

        InitializeRope();
        ValidateAndMaybeClampRopeLength();

        if (snapStraightAtStart && startPointTransform &&
endPointTransform)
        {
            // It snaps initial rope configuration to the straight
line between anchors (no initial sag).
            Vector3 p0 = startPointTransform.position;
            Vector3 p1 = endPointTransform.position;
            for (int i = 0; i <= segmentCount; i++)
            {
                float t = i / (float)segmentCount;
```

```
                Vector3 p = Vector3.Lerp(p0, p1, t);
                ropePoints[i].currentPosition = p;
                ropePoints[i].previousPosition = p;
            }
        }

        // It waits one frame to ensure all Unity components are
    initialized
        yield return null;

        // Optional initial draw
        DrawRope();
    }


    private void InitializeRope()
    // It builds rope geometry, sets initial positions, and
    configures renderers.
    {
        ropePoints = new RopePoint[segmentCount + 1];
        segmentLengths = new float[segmentCount];
        lambdas = new float[segmentCount];
        SegmentTensions = new float[segmentCount];

        // It computes per-point mass and compliance
        RecomputeMaterialAndMass();
        AllocateSolverBuffers();

        Vector3 p0 = startPointTransform ? startPointTransform.
    position : transform.position;
        Vector3 p1 = endPointTransform ? endPointTransform.position :
     p0 + Vector3.right * ropeLengthMeters;

        for (int i = 0; i <= segmentCount; i++)
        {
            float t = i / (float)segmentCount;
            Vector3 p = Vector3.Lerp(p0, p1, t);
            ropePoints[i].currentPosition = p;
            ropePoints[i].previousPosition = p;
            ropePoints[i].locked = (i == 0 && lockFirstPoint) || (i
    == segmentCount && lockLastPoint);
            ropePoints[i].wound = false; // default: not wound
            ropePoints[i].force = Vector3.zero;
        }

        lineRenderer.positionCount = segmentCount + 1;
    }


    private void RecomputeMaterialAndMass()
    /* It computes effective cross-section area, linear mass density,
     Young modulus * area (EA),
     and XPBD compliance. */
    {
        float radius = ropeDiameter * 0.5f;
```

```
        float area = Mathf.PI * radius * radius; // m^2
        linearMass = area * density;             // kg/m
        EA = youngModulus * area;                // N

        segmentRestLength = ropeLengthMeters / segmentCount;

        // Effective compliance for XPBD (scaled if needed with substeps).
        effectiveCompliance = compliance;
        if (scaleComplianceWithSubsteps && substeps > 0)
        {
            // In XPBD, compliance scales with (dt/S)^2
            effectiveCompliance = compliance / (substeps * substeps);
        }

        // For simplicity, it assumes equal mass per free point (anchored/wound points effectively infinite mass).
        float totalMass = linearMass * ropeLengthMeters;
        float freePoints = Mathf.Max(1, segmentCount - (lockFirstPoint ? 1 : 0) - (lockLastPoint ? 1 : 0));
        invMassPerPoint = freePoints > 0 ? freePoints / totalMass : 0f;
    }


    private void AllocateSolverBuffers()
    // It allocates and initializes solver buffers: Lagrange multipliers (lambdas) and per-segment tensions.
    {
        for (int i = 0; i < segmentCount; i++)
        {
            lambdas[i] = 0f;
            SegmentTensions[i] = 0f;
            segmentLengths[i] = segmentRestLength;
        }
    }


    private void Integrate(float dt)
    // Verlet integration step for free (non-anchored, non-wound) points.
    {
        float dt2 = dt * dt;
        for (int i = 0; i <= segmentCount; i++)
        {
            if (ropePoints[i].locked || ropePoints[i].wound) continue;

            Vector3 x = ropePoints[i].currentPosition;
            Vector3 prev = ropePoints[i].previousPosition;

            // Gravity
            Vector3 acc = new Vector3(0f, gravity, 0f);
            // External forces could be added into ropePoints[i].force
```

```

            Vector3 next = x + (x - prev) * damping + acc * dt2 + ropePoints[i].force * dt2 * invMassPerPoint;
            ropePoints[i].previousPosition = x;
            ropePoints[i].currentPosition = next;

            ropePoints[i].force = Vector3.zero;
        }
    }


    private void SolveTimeStep_Tolerance(float dt2)
    // It iterates constraint solving and collisions until the error falls below tolerance or iterations cap.
    {
        int iter = 0;
        float maxErr;
        do
        {
            HandleSegmentCollisions();
            ApplyXPBDConstraints(dt2);
            maxErr = ComputeMaxConstraintError();
            iter++;
        }
        while (iter < xpbdIterations && maxErr > constraintTolerance);

        // Light decay on lambdas for stability
        for (int i = 0; i < segmentCount; i++)
            lambdas[i] *= lambdaDecay;
    }

    private void ApplyXPBDConstraints(float dt2)
    // It applies XPBD distance constraints to enforce segment rest lengths and accumulate lambdas.
    {
        float alpha = effectiveCompliance / dt2; // XPBD parameter

        for (int i = 0; i < segmentCount; i++)
        {
            int a = i;
            int b = i + 1;

            Vector3 pa = ropePoints[a].currentPosition;
            Vector3 pb = ropePoints[b].currentPosition;

            Vector3 delta = pb - pa;
            float dist = delta.magnitude;
            if (dist <= 1e-8f) continue;

            float C = dist - segmentRestLength;
            Vector3 n = delta / dist;

            float wA = (ropePoints[a].locked || ropePoints[a].wound) ? 0f : invMassPerPoint;
```

```
            float wB = (ropePoints[b].locked || ropePoints[b].wound) ? 0f : invMassPerPoint;

            float wSum = wA + wB;
            if (wSum <= 0f) continue;

            // XPBD lambda update
            float denom = wSum + alpha;
            float dlambda = (-C - alpha * lambdas[i]) / denom;
            lambdas[i] += dlambda;

            Vector3 corr = dlambda * n;
            if (wA > 0f) ropePoints[a].currentPosition += corr * (wA / wSum);
            if (wB > 0f) ropePoints[b].currentPosition -= corr * (wB / wSum);

            // Current segment length
            segmentLengths[i] = dist;
        }
    }


    private float ComputeMaxConstraintError()
    // It computes the maximum constraint violation (distance error) across all segments.
    {
        float maxErr = 0f;
        for (int i = 0; i < segmentCount; i++)
        {
            float dist = Vector3.Distance(ropePoints[i].currentPosition, ropePoints[i + 1].currentPosition);
            float err = Mathf.Abs(dist - segmentRestLength);
            if (err > maxErr) maxErr = err;
        }
        return maxErr;
    }


    private void HandleSegmentCollisions()
    // It handles segment collisions by sampling subpoints and projecting them outside colliders.
    {
        if (samplesPerSegment < 1) return;

        for (int i = 0; i < segmentCount; i++)
        {
            Vector3 a = ropePoints[i].currentPosition;
            Vector3 b = ropePoints[i + 1].currentPosition;

            for (int s = 0; s < samplesPerSegment; s++)
            {
                float t = (s + 0.5f) / samplesPerSegment;
                Vector3 p = Vector3.Lerp(a, b, t);

```

```
                // Sphere overlap test; if penetration, it pushes out along normal
                Collider[] hits = Physics.OverlapSphere(p, collisionRadius, collisionMask);
                foreach (var h in hits)
                {
                    if (!h) continue;
                    Vector3 closest = h.ClosestPoint(p);
                    Vector3 dir = p - closest;
                    float d = dir.magnitude;
                    if (d < collisionRadius && d > 1e-6f)
                    {
                        Vector3 n = dir / d;
                        Vector3 correction = n * (collisionRadius - d);

                        // It distributes correction to the two endpoints proportionally
                        float wA = (ropePoints[i].locked || ropePoints[i].wound) ? 0f : 1f;
                        float wB = (ropePoints[i + 1].locked || ropePoints[i + 1].wound) ? 0f : 1f;
                        float wSum = wA + wB;

                        if (wSum > 0f)
                        {
                            ropePoints[i].currentPosition += correction * (wA / wSum) * (1f - t);
                            ropePoints[i + 1].currentPosition += correction * (wB / wSum) * t;
                        }
                    }
                }
            }
        }
    }


    private void ComputeTensionFromLambdas(float dt2)
    // It recovers segment tensions from Lagrange multipliers after XPBD.
    {
        float alpha = effectiveCompliance / dt2;
        float Tmax = 0f;

        for (int i = 0; i < segmentCount; i++)
        {
            float T = Mathf.Abs(lambdas[i]) / (Mathf.Sqrt(dt2) + alpha);
            SegmentTensions[i] = T;
            if (T > Tmax) Tmax = T;
        }
        Tension = Tmax;
    }

```

```

    private void ApplyBackReactionForces()
    // It applies equal and opposite reaction forces to the endpoints (rigidbodies).
    {
        if (!floaterRb && !spoolRb) return;

        // It approximates total rope force along the end segments
        Vector3 fStart = Vector3.zero;
        Vector3 fEnd = Vector3.zero;

        if (segmentCount >= 1)
        {
            Vector3 d0 = ropePoints[1].currentPosition - ropePoints[0].currentPosition;
            Vector3 d1 = ropePoints[segmentCount].currentPosition - ropePoints[segmentCount - 1].currentPosition;

            float len0 = d0.magnitude;
            float len1 = d1.magnitude;

            if (len0 > 1e-6f) fStart = (d0 / len0) * SegmentTensions[0];
            if (len1 > 1e-6f) fEnd = -(d1 / len1) * SegmentTensions[segmentCount - 1];
        }

        if (floaterRb) floaterRb.AddForce(fStart, ForceMode.Force);
        if (spoolRb) spoolRb.AddForce(fEnd, ForceMode.Force);
    }


    private void DrawRope()
    // It draws the rope polyline through the current point positions using a LineRenderer.
    {
        for (int i = 0; i <= segmentCount; i++)
            lineRenderer.SetPosition(i, ropePoints[i].currentPosition);
    }


    private void ValidateAndMaybeClampRopeLength()
    // It ensures ropeLengthMeters is not shorter than the anchor distance; optionally clamp.
    {
        if (!(startPointTransform && endPointTransform)) return;

        float d = Vector3.Distance(startPointTransform.position, endPointTransform.position);

        if (clampRopeLengthToAnchors && ropeLengthMeters < d)
        {
```

```
            Debug.LogWarning($"[Rope] ropeLengthMeters ({ropeLengthMeters:F4} m) < anchor distance ({d:F4} m). Clamp at distance.");
            ropeLengthMeters = d;
        }
        else if (ropeLengthMeters < d)
        {
            Debug.LogWarning($"[Rope] ropeLengthMeters ({ropeLengthMeters:F4} m) < anchor distance ({d:F4} m). Impossible state: increase the length or bring the anchors closer.");
        }
    }


    private float MaxSagFromChord()
    // It estimates the maximum sag based on chord length and slack (approximate catenary).
    {
        if (!(startPointTransform && endPointTransform)) return 0f;

        // Chord length (straight line between anchors)
        float d = Vector3.Distance(startPointTransform.position, endPointTransform.position);
        float L = ropeLengthMeters;
        if (L <= d) return 0f;

        float slack = L - d;

        // Simple approximation: sag = k * sqrt(slack * d) with tuned factor
        float sag = Mathf.Sqrt(Mathf.Max(0f, slack * d)) * 0.5f;
        return sag;
    }


    private void SanityCheck()
    // It performs consistency checks (arc length vs. target, sag limits, constraint error) and log summaries.
    {
        if (!enableSanityCheck) return;

        frameCounter++;
        if (frameCounter % logEveryNFrames != 0) return;

        // 1) Arc length vs. target length
        float arc = CurrentArcLength();
        float d = Vector3.Distance(startPointTransform.position, endPointTransform.position);
        float slack = Mathf.Max(0f, ropeLengthMeters - d);
        float maxSag = MaxSagFromChord();

        if (Mathf.Abs(arc - ropeLengthMeters) > Mathf.Max(arcTolAbs, arcTolRel * ropeLengthMeters))
        {
```

```
            Debug.LogWarning($"[Rope] Arc!=L: arc={arc:F6} m differs by {arc - ropeLengthMeters:+0.000000;-0.000000} m, tolAbs={arcTolAbs:E2}, tolRel={arcTolRel:P2}");
        }

        // 2) Segment sum vs rope target length
        float segSum = 0f;
        for (int i = 0; i < segmentCount; i++)
            segSum += Vector3.Distance(ropePoints[i].currentPosition, ropePoints[i + 1].currentPosition);

        if (Mathf.Abs(segSum - ropeLengthMeters) > Mathf.Max(arcTolAbs, arcTolRel * ropeLengthMeters))
        {
            Debug.LogWarning($"[Rope] Inconsistent: segmentLengthSum={segSum:F6} m different from ropeLengthMeters={ropeLengthMeters:F6} m");
        }

        // 3) Excessive sag vs slack (heuristic)
        float sagLim = Mathf.Max(0.001f, maxSagFraction * ropeLengthMeters);
        if (maxSag > Mathf.Max(sagLim, sagVsSlackFactor * slack))
        {
            Debug.LogWarning($"[Rope] Excessive sag: maxSag={maxSag:F4} m exceeds limit {sagLim:F4} m, slack={slack:F4} m. MaxTension={Tension:F2} N");
        }

        // 4) High constraint error -> use a higher factor for the warning.
        if (warnOnHighConstraintError)
        {
            float maxErr = ComputeMaxConstraintError();
            if (maxErr > warnConstraintFactor * constraintTolerance)
                Debug.LogWarning($"[Rope] Constraints error: {maxErr:E3} m (>{warnConstraintFactor}xtol). Increase iterations/substeps or rigidity (E).");
        }

        // Compact informational log
        Debug.Log($"[Rope] d={d:F4}  L={ropeLengthMeters:F4}  arc={arc:F4}  slack={slack:F4}  sag={maxSag:F4}  Tmax={Tension:F2}N");
    }


    public float CurrentArcLength()
    // It computes the current rope arc length by summing segment distances.
    {
        float L = 0f;
        for (int i = 0; i < segmentCount; i++)
            L += Vector3.Distance(ropePoints[i].currentPosition, ropePoints[i + 1].currentPosition);
        return L;
```

```
        }
}
```

Appendix C

PIDTensionController.cs - Deployed case

```
/*
 * Project    : A digital twin of an air-bearing platform for tethered satellite systems:
                 from tether deployment to post-deployment control
 * File       : PIDTensionController.cs
 * Author     : Edoardo De Blasi
 * Supervisors: Prof. Paolo Maggiore, Dr. Giuseppe Governale, Prof. Stephanie Lizy-Destrez
 * Date       : September 2025
 * Notes      : Developed in Unity using C# scripting.
 * License    : This code is intended for academic and research purposes only.
*/

using UnityEngine;
using System.IO;
/// <summary>
/// PID tension controller component.
/// Computes roll/pitch commands for the floater each frame.
/// </summary>

public class PIDTensionController : FloaterController
{
    [Header("Scene Objects")]
    public VerletRope tetherRope;
    public Rigidbody floater;
    public Transform bench;
    public Transform tetherAnchor;

    [Header("PID Control Settings")]
    public float desiredTension = 50f;
    public float Kp = 1.0f;
    public float Ki = 0.1f;
    public float Kd = 0.2f;

    [Header("Scaling")]
    public bool applyScaling = true;
```

```
    public enum ScalingScenario { HCW_Emulation, Tether_Deployed }
    public ScalingScenario scenario = ScalingScenario.Tether_Deployed;
    [Tooltip("Override lambdaL and lambdat. Leave 0 to auto-fill from scenario.")]
    public float lambdaL = 0f;  // length scale (real/table)
    public float lambdaT = 0f;  // time scale (real/table)

    [Header("Simulation Settings")]
    public float maxAngleDeg = 3.0f;

    [Header("PID Safeguards & Filters")]
    [Tooltip("Clamp for the integral term to prevent windup (in tension units x s)")]
    public float integralClamp = 1000f;
    [Tooltip("Simple low-pass filter cutoff (Hz) applied to measured tension to reduce noise")]
    public float tensionLPFCutoffHz = 3f;
    [Tooltip("Optional clamp on commanded acceleration magnitude (m/s^2)")]
    public float outputAccClamp = 5f;

    private float filteredTension = 0f;

    private float integral = 0f;
    private float previousError = 0f;

    // CSV logging
    private StreamWriter csvWriter;
    public string fileName = "FloaterTensionData.csv";
    private string filePath;

    [Header("Actuator Simulation")]
    public float actuatorSmoothTime = 0.2f; // Time (in seconds) to reach the target: lower = faster
    // It simulates the actuators inertia

    // SmoothDamp variables
    private float currentRoll, currentPitch;
    private float rollVelocity, pitchVelocity;

    void Start()
    {
        if (applyScaling)
        {
            if (lambdaL <= 0f || lambdaT <= 0f)
            {
                if (scenario == ScalingScenario.HCW_Emulation)
                {
                    // Scaling factors with only HCW equations, no tether
                    lambdaL = 700f;
                    lambdaT = 500f;
                }
                else
```

```
                {
                    // In case of tethered floater
                    lambdaL = 50f;
                    lambdaT = 20f;
                }
            }
        }

        if (tetherRope != null) filteredTension = tetherRope.Tension;

        // It sets CSV file path in the specified folder
        string folderPath = "/Users/edoardodeblasi/Desktop/Documenti vari/Università/Tesi Magistrale/Digital twin/Risultati tensione";

        // It checks if the folder exists; if not, creates it
        if (!Directory.Exists(folderPath))
        {
            Directory.CreateDirectory(folderPath);
            Debug.Log("Folder created: " + folderPath);
        }

        filePath = Path.Combine(folderPath, fileName);

        try
        {
            FileStream fileStream = new FileStream(filePath, FileMode.Create, FileAccess.Write, FileShare.ReadWrite);
            csvWriter = new StreamWriter(fileStream);
            csvWriter.WriteLine("Time,PosX,PosY,PosZ,Tension");
            Debug.Log("CSV created at: " + filePath);
        }
        catch (System.Exception e)
        {
            Debug.LogError("CSV file open error: " + e.Message);
            enabled = false;
        }
    }
    /// <summary>
    /// Main control step: reads current state, computes desired acceleration,
    /// maps acceleration to small roll/pitch angles, and applies smoothed commands.
    /// </summary>

    public override void ComputeControl()
    {
        if (tetherRope == null || floater == null || bench == null || tetherAnchor == null) return;

        // It uses the actual dt of the current frame/step
        float dt = Time.inFixedTimeStep ? Time.fixedDeltaTime : Time.deltaTime;
        if (dt <= 0f) return;

```

```
        // Exponential low-pass filter on the tension measurement (reduces noise/crazy derivative)
        float cutoff = Mathf.Max(0f, tensionLPFCutoffHz);
        if (cutoff > 0f)
        {
            float alpha = Mathf.Exp(-2f * Mathf.PI * cutoff * dt); // 0..1, closer to 0 = faster filter
            filteredTension = filteredTension + (1f - alpha) * (tetherRope.Tension - filteredTension);
        }
        else
        {
            filteredTension = tetherRope.Tension;
        }

        // PID on tension error
        float error = desiredTension - filteredTension;

        // Integrator with anti-windup (clamp)
        integral += error * dt;
        if (integralClamp > 0f)
            integral = Mathf.Clamp(integral, -integralClamp, integralClamp);

        float derivative = (error - previousError) / dt;
        previousError = error;

        // Control acceleration computation
        float outputAcceleration = Kp * error + Ki * integral + Kd * derivative;

        // Direction: from the anchor point to the floater
        Vector3 directionToPush = floater.position - tetherAnchor.position;
        directionToPush.y = 0f;
        if (directionToPush.sqrMagnitude > 1e-9f)
            directionToPush.Normalize();
        else
            directionToPush = Vector3.zero;

        Vector3 controlAcceleration = directionToPush * outputAcceleration;

        // Scaling for emulation (a_table = (lambdat^2/lambdaL) * a_real)
        if (applyScaling)
        {
            float scaleAcc = (lambdaT * lambdaT) / Mathf.Max(1e-6f, lambdaL);
            controlAcceleration *= scaleAcc;
        }

        // Optional clamp on commanded acceleration to avoid extreme saturation
        if (outputAccClamp > 0f)
```

```
        {
            float mag = controlAcceleration.magnitude;
            if (mag > outputAccClamp)
            {
                controlAcceleration = controlAcceleration * (outputAccClamp / mag);
            }
        }

        float g = 9.81f;
        float maxAngleRad = Mathf.Deg2Rad * Mathf.Max(0.1f, maxAngleDeg);

        // Conversion of the control acceleration into angles (in radians)
        // small angle: a = g * theta
        float targetRollRad  = Mathf.Clamp( controlAcceleration.z / g, -maxAngleRad, maxAngleRad);
        float targetPitchRad = Mathf.Clamp(-controlAcceleration.x / g, -maxAngleRad, maxAngleRad);

        // Basic anti-windup
        {
            float desiredRollRad  =  controlAcceleration.z / g;
            float desiredPitchRad = -controlAcceleration.x / g;
            float satErr = (desiredRollRad - targetRollRad) + (desiredPitchRad - targetPitchRad);
            // small coefficient to avoid destabilizing (tuning):
            integral -= 0.1f * satErr * dt;
        }

        // Actuator smoothing working in degrees to use SmoothDampAngle
        float targetRollDeg  = targetRollRad  * Mathf.Rad2Deg;
        float targetPitchDeg = targetPitchRad * Mathf.Rad2Deg;

        float currentRollDeg  = currentRoll  * Mathf.Rad2Deg;
        float currentPitchDeg = currentPitch * Mathf.Rad2Deg;

        currentRoll  = Mathf.SmoothDampAngle(currentRollDeg, targetRollDeg,  ref rollVelocity,  actuatorSmoothTime) * Mathf.Deg2Rad;
        currentPitch = Mathf.SmoothDampAngle(currentPitchDeg, targetPitchDeg, ref pitchVelocity, actuatorSmoothTime) * Mathf.Deg2Rad;

        // It applies the smoothed rotation
        bench.rotation = Quaternion.Euler(currentRoll * Mathf.Rad2Deg, 0f, currentPitch * Mathf.Rad2Deg);

        // Log CSV
        if (csvWriter != null)
        {
            Vector3 pos = floater.position;
```

```
                csvWriter.WriteLine($"{Time.time:F3},{pos.x:F4},{pos.y:F4
    },{pos.z:F4},{tetherRope.Tension:F4}");
            }
        }

        void OnApplicationQuit()
        {
            if (csvWriter != null)
            {
                csvWriter.Flush();
                csvWriter.Close();
                Debug.Log("CSV saved at: " + filePath);
            }
        }
}
```

Appendix D

PIDPositionController.cs - Deployed case

```
/*
 * Project    : A digital twin of an air-bearing platform for tethered satellite systems:
                from tether deployment to post-deployment control
 * File       : PIDPositionController.cs
 * Author     : Edoardo De Blasi
 * Supervisors: Prof. Paolo Maggiore, Dr. Giuseppe Governale, Prof. Stephanie Lizy-Destrez
 * Date       : September 2025
 * Notes      : Developed in Unity using C# scripting.
 * License    : This code is intended for academic and research purposes only.
*/
using UnityEngine;
using System.IO;
/// <summary>
/// PID position controller component.
/// Computes roll/pitch commands for the floater each frame.
/// </summary>

public class PIDPositionController : FloaterController
{
    [Header("Scene Objects")]
    public VerletRope tetherRope;
    public Rigidbody floater;
    public Transform bench;
    public Transform tetherAnchor;

    [Header("Tether Properties")]
    public float tetherRestLength = 5.0f;
    public float tetherStiffness = 100f;

    [Header("PID Control Settings")]
    public Vector3 desiredPos = new Vector3(-5f, 0f, 1.7f);
    public float Kp = 100f;
    public float Ki = 0f;
    public float Kd = 20f;
```

```

    [Header("Scaling (per paper)")]
    public bool applyScaling = true;
    public enum ScalingScenario { HCW_Emulation, Tether_Deployed }
    public ScalingScenario scenario = ScalingScenario.HCW_Emulation;
    [Tooltip("If true, desiredPos is given in REAL (orbital) meters and will be divided by lambdaL to map onto the table. If false, desiredPos is already in table meters.")]
    public bool desiredPosIsReal = false;

    [Tooltip("Override lambdaL and lambdat. Leave 0 to auto-fill from scenario.")]
    public float lambdaL = 0f;  // length scale (real/table)
    public float lambdaT = 0f;  // time scale (real/table)

    [Header("Simulation Settings")]
    public float maxAngleDeg = 2.0f;

    private Vector3 integral;
    private Vector3 previousError;
    private float deltaT;

    // CSV logging
    private StreamWriter csvWriter;
    public string fileName = "FloaterTensionData.csv";
    private string filePath;

    [Header("Actuator Simulation")]
    public float actuatorSmoothTime = 0.2f; // Time (in seconds) to reach the target: lower = faster
    // It simulates the actuators inertia

    // SmoothDamp variables
    private float currentRoll, currentPitch;
    private float rollVelocity, pitchVelocity;

    void Start()
    {
        deltaT = Time.fixedDeltaTime;
        if (deltaT == 0)
        {
            Debug.LogError("Time.fixedDeltaTime is zero. Ensure you are in a FixedUpdate context.");
            enabled = false;
            return;
        }

        if (applyScaling)
        {
            if (lambdaL <= 0f || lambdaT <= 0f)
            {
                if (scenario == ScalingScenario.HCW_Emulation)
                {
                    // Scaling factors with only HCW equations, no tether
```

```
                    lambdaL = 700f;
                    lambdaT = 500f;
                }
                else
                {
                    // In case of tethered floater
                    lambdaL = 50f;
                    lambdaT = 20f;
                }
            }
            if (desiredPosIsReal)
            {
                // Conversion from real coordinates to table coordinates
                desiredPos /= lambdaL;
            }
        }

        // It sets CSV file path in the specified folder
        string folderPath = "/Users/edoardodeblasi/Desktop/Documenti vari/Università/Tesi Magistrale/Digital twin/Risultati tensione";

        // It checks if the folder exists; if not, creates it
        if (!Directory.Exists(folderPath))
        {
            Directory.CreateDirectory(folderPath);
            Debug.Log("Folder created: " + folderPath);
        }

        filePath = Path.Combine(folderPath, fileName);

        try
        {
            FileStream fileStream = new FileStream(filePath, FileMode.Create, FileAccess.Write, FileShare.ReadWrite);
            csvWriter = new StreamWriter(fileStream);
            csvWriter.WriteLine("Time,PosX,PosY,PosZ,Tension");
            Debug.Log("CSV created at: " + filePath);
        }
        catch (System.Exception e)
        {
            Debug.LogError("CSV file open error: " + e.Message);
            enabled = false;
        }
    }
    /// <summary>
    /// Main control step: reads current state, computes desired acceleration,
    /// maps acceleration to small roll/pitch angles, and applies smoothed commands.
    /// </summary>

    public override void ComputeControl()
    {
        Vector3 pos = floater.position;
```

```
        Vector3 velocity = floater.linearVelocity;

        // Errors computation
        Vector3 error = desiredPos - pos;
        integral += error * deltaT;
        Vector3 derivative = (error - previousError) / deltaT;

        previousError = error;
        // Control acceleration computation
        Vector3 controlAcceleration = Kp * error + Ki * integral + Kd * derivative;
        if (applyScaling)
        {
            float scaleAcc = (lambdaT * lambdaT) / Mathf.Max(1e-6f, lambdaL);
            controlAcceleration *= scaleAcc;
        }

        // Conversion of the control acceleration into angles (in radians)
        // small angle: a = g * theta
        float g = 9.81f;
        float maxAngleRad = Mathf.Deg2Rad * maxAngleDeg;
        float targetPitchRad = Mathf.Clamp(-controlAcceleration.x / g, -maxAngleRad, maxAngleRad);
        float targetRollRad = Mathf.Clamp(controlAcceleration.z / g, -maxAngleRad, maxAngleRad);

        // It applies SmoothDamp to the angles (works in degrees)
        float currentRollDeg = currentRoll * Mathf.Rad2Deg;
        float currentPitchDeg = currentPitch * Mathf.Rad2Deg;

        currentRoll = Mathf.SmoothDampAngle(currentRollDeg, targetRollRad * Mathf.Rad2Deg, ref rollVelocity, actuatorSmoothTime) * Mathf.Deg2Rad;
        currentPitch = Mathf.SmoothDampAngle(currentPitchDeg, targetPitchRad * Mathf.Rad2Deg, ref pitchVelocity, actuatorSmoothTime) * Mathf.Deg2Rad;

        // It applies the smoothed rotation
        bench.rotation = Quaternion.Euler(currentRoll * Mathf.Rad2Deg, 0f, currentPitch * Mathf.Rad2Deg);

        // It writes data to CSV
        if (csvWriter != null)
        {
            csvWriter.WriteLine($"{Time.time:F3},{floater.position.x:F4},{floater.position.y:F4},{floater.position.z:F4},{tetherRope.Tension:F4}");
        }
    }

    void OnApplicationQuit()
    {
        if (csvWriter != null)
```

```
            {
                csvWriter.Close();
                Debug.Log("CSV saved at: " + filePath);
            }
        }

}
```

Appendix E

LQRPositionController.cs - Deployed case

```
/*
 * Project    : A digital twin of an air-bearing platform for tethered
    satellite systems:
                 from tether deployment to post-deployment control
 * File       : LQRPositionController.cs
 * Author     : Edoardo De Blasi
 * Supervisors: Prof. Paolo Maggiore, Dr. Giuseppe Governale, Prof.
   Stephanie Lizy-Destrez
 * Date       : September 2025
 * Notes      : Developed in Unity using C# scripting.
 * License    : This code is intended for academic and research
   purposes only.
*/
using UnityEngine;
using System.IO;
using MathNet.Numerics.LinearAlgebra;
using MathNet.Numerics.LinearAlgebra.Double;
/// <summary>
/// LQR position controller component.
/// Computes roll/pitch commands for the floater each frame.
/// </summary>

public class LQRPositionController : FloaterController
{
    [Header("Scene Objects")]
    public VerletRope tetherRope;
    public Rigidbody floater;
    public Transform bench;
    public Transform tetherAnchor;

    [Header("Tether Properties")]
    public float tetherRestLength = 5.0f;
    public float tetherStiffness = 100f;

    [Header("LQR Control Settings")]
    public Vector3 desiredPos = new Vector3(-5f, 0f, 1.7f);
    public float omega = 0.1f;
```

```
    private Vector3 previousPosition;
    private Vector3 velocity;
    private float deltaT;
    private Matrix<double> K;

    [Header("Scaling (per paper)")]
    public bool applyScaling = true;
    public enum ScalingScenario { HCW_Emulation, Tether_Deployed }
    public ScalingScenario scenario = ScalingScenario.HCW_Emulation;
    [Tooltip("If true, desiredPos is given in REAL (orbital) meters
   and will be divided by lambdaL to map onto the table. If false,
   desiredPos is already in table meters.")]
    public bool desiredPosIsReal = false;

    [Tooltip("Override lambdaL and lambdat. Leave 0 to auto-fill from
    scenario.")]
    public float lambdaL = 0f;  // length scale (real/table)
    public float lambdaT = 0f;  // time scale (real/table)

    [Tooltip("If you provide the real mean motion Omega [rad/s],
   enabling this will compute omega = Omega * lambdat as in the paper
   . If disabled, the public field 'omega' is used as-is.")]
    public bool useOmegaFromReal = false;
    public float realOmega = 0f; // Omega [rad/s]

    [Header("Simulation Settings")]
    public float maxAngleDeg = 2.0f;

    [Header("Q (4x4) - Diagonal Values")]
    public float q11 = 1000f;
    public float q22 = 1000f;
    public float q33 = 100f;
    public float q44 = 100f;

    [Header("R (2x2) - Diagonal Values")]
    public float r11 = 1000f;
    public float r22 = 1000f;


    // CSV logging
    private StreamWriter csvWriter;
    public string fileName = "FloaterTensionData.csv";
    private string filePath;

    [Header("Actuator Simulation")]
    public float actuatorSmoothTime = 0.2f; // Time (in seconds) to
   reach the target: lower = faster
    // It simulates the actuators inertia

    // SmoothDamp variables
    private float currentRoll, currentPitch;
    private float rollVelocity, pitchVelocity;

    void Start()
    {
```

```
        deltaT = Time.fixedDeltaTime;
        if (deltaT == 0)
        {
            Debug.LogError("Time.fixedDeltaTime is zero. Ensure you are in a FixedUpdate context.");
            enabled = false;
            return;
        }

        // --- Scaling factors definition ---
        if (applyScaling)
        {
            if (lambdaL <= 0f || lambdaT <= 0f)
            {
                if (scenario == ScalingScenario.HCW_Emulation)
                {
                    // Scaling factors with only HCW equations, no tether
                    lambdaL = 700f;  // real/table
                    lambdaT = 500f;  // real/table
                }
                else
                {
                    // In case of tethered floater
                    lambdaL = 50f;
                    lambdaT = 20f;
                }
            }
            if (useOmegaFromReal && realOmega > 0f)
            {
                // Omega scales with time: omega = RealOmega * lambdat
                omega = realOmega * lambdaT;
            }
            if (desiredPosIsReal)
            {
                // Conversion from real coordinates to table coordinates
                desiredPos /= lambdaL;
            }
        }

        var A = DenseMatrix.OfArray(new double[,] {
            { 0, 0, 1, 0 },
            { 0, 0, 0, 1 },
            { 3 * omega * omega, 0, 0, 2 * omega },
            { 0, 0, -2 * omega, 0 }
        });

        var B = DenseMatrix.OfArray(new double[,] {
            { 0, 0 },
            { 0, 0 },
            { 1, 0 },
            { 0, 1 }
        });
```

```

        // Creation of the Q and R matrices from the inserted diagonal values
        var Q = DenseMatrix.OfDiagonalArray(new double[] { q11, q22, q33, q44 });
        var R = DenseMatrix.OfDiagonalArray(new double[] { r11, r22 });

        var P = SolveCARE(A, B, Q, R);
        K = R.Inverse() * B.TransposeThisAndMultiply(P); // Computation of the gain matrix K

        Debug.Log("Computed K matrix (LQR):\n" + K.ToString());

        previousPosition = floater.position;

        // It sets CSV file path in the specified folder
        string folderPath = "/Users/edoardodeblasi/Desktop/Documenti vari/Università/Tesi Magistrale/Digital twin/Risultati tensione";

        // It checks if the folder exists; if not, creates it
        if (!Directory.Exists(folderPath))
        {
            Directory.CreateDirectory(folderPath);
            Debug.Log("Folder created: " + folderPath);
        }

        filePath = Path.Combine(folderPath, fileName);

        try
        {
            //FileStream fileStream = new FileStream(filePath, FileMode.Create, FileAccess.Write, FileShare.ReadWrite);
            csvWriter = new StreamWriter(filePath);
            csvWriter.WriteLine("Time,PosX,PosY,PosZ,Tension");
            Debug.Log("CSV created at: " + filePath);
        }
        catch (System.Exception e)
        {
            Debug.LogError("CSV file open error: " + e.Message);
            enabled = false;
        }
    }
    /// <summary>
    /// Main control step: reads current state, computes desired acceleration,
    /// maps acceleration to small roll/pitch angles, and applies smoothed commands.
    /// </summary>


    public override void ComputeControl()
    {
        Vector3 pos = floater.position;
        Vector3 velocity = floater.linearVelocity;
```

```

        // State/error vector definition
        Vector<double> stateError = DenseVector.OfArray(new double[] {
            desiredPos.x - pos.x,
            desiredPos.z - pos.z,
            velocity.x,
            velocity.z
        });

        // Control acceleration computation
        Vector<double> controlAcceleration = -K * stateError;
        float accX = (float)controlAcceleration[0];
        float accZ = (float)controlAcceleration[1];
        if (applyScaling)
        {
            // a_table = (lambdat^2/lambdaL) * a_real
            float scaleAcc = (lambdaT * lambdaT) / Mathf.Max(1e-6f, lambdaL);
            accX *= scaleAcc;
            accZ *= scaleAcc;
        }

        // Conversion of the control acceleration into angles (in radians)
        // small angle: a = g * theta
        float g = 9.81f;
        float maxAngleRad = Mathf.Deg2Rad * maxAngleDeg;
        float targetRollRad = Mathf.Clamp(accZ / g, -maxAngleRad, maxAngleRad);
        float targetPitchRad = Mathf.Clamp(accX / g, -maxAngleRad, maxAngleRad);


        // It applies SmoothDamp to the angles (works in degrees)
        float currentRollDeg = currentRoll * Mathf.Rad2Deg;
        float currentPitchDeg = currentPitch * Mathf.Rad2Deg;

        currentRoll = Mathf.SmoothDampAngle(currentRollDeg, targetRollRad * Mathf.Rad2Deg, ref rollVelocity, actuatorSmoothTime) * Mathf.Deg2Rad;
        currentPitch = Mathf.SmoothDampAngle(currentPitchDeg, targetPitchRad * Mathf.Rad2Deg, ref pitchVelocity, actuatorSmoothTime) * Mathf.Deg2Rad;

        // It applies the smoothed rotation
        bench.rotation = Quaternion.Euler(currentRoll * Mathf.Rad2Deg, 0f, -currentPitch * Mathf.Rad2Deg);

        // It writes data to CSV
        if (csvWriter != null)
        {
            csvWriter.WriteLine($"{Time.time:F3},{floater.position.x:F4},{floater.position.y:F4},{floater.position.z:F4},{tetherRope.Tension:F4}");
```

```
        }
    }

        Matrix<double> SolveCARE(Matrix<double> A, Matrix<double> B,
    Matrix<double> Q, Matrix<double> R)
    {
        var Rinv = R.Inverse();
        var P = Q.Clone();

        for (int i = 0; i < 100; i++)
        {
            var Pdot = A.TransposeThisAndMultiply(P) + P * A - P * B
    * Rinv * B.TransposeThisAndMultiply(P) + Q;
            P -= Pdot * 0.001;
        }

        return P;
    }

    void OnApplicationQuit()
    {
        if (csvWriter != null)
        {
            csvWriter.Close();
            Debug.Log("CSV saved at: " + filePath);
        }
    }
}
```

Appendix F

XPBDTetherWithSpool.cs - Deployment

```csharp
/*
 * Project    : A digital twin of an air-bearing platform for tethered
    satellite systems:
                from tether deployment to post-deployment control
 * File       : XPBDTetherWithSpool.cs - Deployment case
 * Author     : Edoardo De Blasi
 * Supervisors: Prof. Paolo Maggiore, Dr. Giuseppe Governale, Prof.
   Stephanie Lizy-Destrez
 * Date       : September 2025
 * Notes      : Developed in Unity using C# scripting.
 * License    : This code is intended for academic and research
   purposes only.
*/
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Globalization;
using System.Text;

[DefaultExecutionOrder(10000)]
[RequireComponent(typeof(LineRenderer))]

/// <summary>
/// XPBD-based tether/rope with optional kinematic spool deployment,
   simple collisions, and CSV logging.
/// </summary>
public class XPBDTetherWithSpool : MonoBehaviour
{
    // -------- Anchors --------
    [Header("Anchors")]
    public Transform startPointTransform; // spool side
    public Transform endPointTransform;   // floater side

    // === Back-reaction & COM ===
    [Header("Back-reaction & Floater COM")]
    [SerializeField] private bool applyBackReaction = true;
```

```
    [SerializeField] private bool autoFindRigidbodies = true;
    public Rigidbody startRB;  // spool
    public Rigidbody endRB; // floater
    [SerializeField] private bool setEndRbCenterOfMass = true;
    [SerializeField] private Vector3 endRbCenterOfMassLocalOffset = new Vector3(0f, -0.2f, 0f); // lowers the CoM
    [SerializeField] private bool setStartRbCenterOfMass = false;
    [SerializeField] private Vector3 startRbCenterOfMassLocalOffset = Vector3.zero;

    // -------- Spool Deployment --------
    [Header("Spool Deployment")]
    [Tooltip("Cylinder (spool) Transform. Required for kinematic deployment.")]
    public Transform spoolTransform;
    [Tooltip("Spool RigidBody (optional). Used to stop the spool at the end of deployment.")]
    public Rigidbody spoolRB;

    [Tooltip("Effective radius on which the tether adheres while it is wound.")]
    public float spoolRadius = 0.15f;

    [Tooltip("Helical pitch per turn (m) - adjustable in the Inspector.")]
    public float helixPitch = 0.02f;

    [Tooltip("If true, the code rotates the spool kinematically during deployment.")]
    public bool driveSpoolRotation = true;

    [Tooltip("Angular velocity (rad/s) when driven by code.")]
    public float spoolAngularSpeed = 2.5f;

    [Tooltip("Angular window around the 'lowest point' to release a segment (degrees).")]
    [Range(2f, 45f)] public float releaseWindowDeg = 10f;

    [Tooltip("Extra margin to avoid pre-tension in the estimate of the excess length.")]
    public float extraSlackMargin = 0.02f;

    [Tooltip("Always pin point 0 on the spool; release all others.")]
    public bool keepFirstPointAttached = true;

    // -------- Tether --------
    [Header("Tether Settings")]
    [SerializeField, Min(2)] private int segmentCount = 35;

    [Tooltip("Tether length")]
    [SerializeField, Min(1e-4f)] private float ropeLengthMeters = 1f;

    [Tooltip("If false, it assigns the tether a length equal to the distance between the anchors")]
    [SerializeField] private bool useInspectorRopeLength = true;
```

```

    [Tooltip("If true and the anchors are farther apart than ropeLength, adjust ropeLength to the distance to avoid an impossible state")]
    [SerializeField] private bool clampRopeLengthToAnchors = true;

    [SerializeField] private bool snapStraightAtStart = true;
    [SerializeField] private float gravity = -9.81f;
    [SerializeField, Range(0.90f, 1f)] private float damping = 0.997f;

    // -------- Material --------
    [Header("Material (Hooke)")]
    [SerializeField] private float youngModulus = 8.0e9f;   // Pa
    [SerializeField] private float ropeDiameter = 0.00016f; // m
    [SerializeField] private float density = 1540f;         // kg/m^3

    // -------- Collisions --------
    [Header("Collisions")]
    [SerializeField] private bool handleCollisions = true;
    [SerializeField] private LayerMask collisionLayers = ~0;

    // -------- Solver --------
    [Header("Solver")]
    [SerializeField, Range(1, 8)] private int solverSubsteps = 4;  // substeps
    [SerializeField, Min(1)] private int iterations = 180;  // max iterations per substep
    [SerializeField] private float constraintTolerance = 1e-5f;  // max error (m) per substep
    [SerializeField, Range(1f, 1.99f)] private float sor = 1.85f;  // over-relaxation (Gauss-Seidel)
    [SerializeField] private bool warmstart = true;  // Does not reset lambda between substeps
    [SerializeField, Range(0.9f, 1f)] private float lambdaDecay = 0.98f; // lambda decay each frame
    [SerializeField] private bool scaleComplianceWithSubsteps = true;  // c' = c / S^2

    // -------- CSV Logging (NEW) --------
    [Header("CSV Logging")]
    [Tooltip("Abilita il salvataggio su CSV di Time,PosX,PosY,PosZ,Tension.")]
    public bool enableCsvLogging = true;

    [Tooltip("Cartella di destinazione del CSV.")]
    public string csvDirectory = "'/Users/edoardodeblasi/Desktop/Documenti vari/Università/Tesi Magistrale/Digital twin/Risultati";

    [Tooltip("Base file name (without extension). The file will be <Name>.csv")]
    public string csvBaseFileName = "FloaterTensionData";

    [Tooltip("If true, it appends to the existing file; otherwise, it overwrites it by writing the header.")]
```

```

    private string csvFullPath;
    private StreamWriter csvWriter;

    // -------- Sanity check --------
    [Header("Sanity Check (log)")]
    [SerializeField] private float arcTolAbs = 1e-3f;      // 1 mm
    [SerializeField] private float arcTolRel = 2e-3f;      // 0.2% of L
    [SerializeField] private float sagVsSlackFactor = 3.5f;
    [SerializeField] private float warnConstraintFactor = 20f;
    [SerializeField] private bool enableSanityCheck = true;
    [SerializeField, Min(1)] private int logEveryNFrames = 30;

    [Tooltip("Afflosciamento massimo ammesso (frazione della lunghezza totale).")]
    [SerializeField, Range(0.01f, 0.75f)] private float maxSagFraction = 0.35f;

    [Tooltip("If true, logs a warning when the average constraint error is high for multiple consecutive substeps.")]
    [SerializeField] private bool warnOnHighConstraintError = true;

    [Header("Output")]
    [HideInInspector] public float Tension { get; private set; }
    [HideInInspector] public float[] SegmentTensions { get; private set; }


    private LineRenderer lineRenderer;

    private struct RopePoint
    {
        public Vector3 currentPosition;
        public Vector3 previousPosition;
        public bool isLocked; // true if the point is constrained (anchor or spool)
    }

    private readonly List<RopePoint> ropePoints = new List<RopePoint>();
    private float segmentLength;
    private float[] lambdas;          // XPBD multipliers
    private float[] compliance;       // c per segment
    private float[] invMass;          // 1/m per point
    private float crossSectionArea;   // A = pi r^2
    private float linearDensity;      // mu = A * rho
    private float lastDt2Sub = 0f;    // ComputeTension
    private int _frameCount = 0;

    // ---- Deployment state ----
    private int wrappedPointCount = 0;      // how many points are still wound on the spool
    private bool deploymentActive = false;
    private bool deploymentComplete = false;
```

```
    private float angleOffset = 0f;          // angular offset
    private float helixTurnLength = 0f;      // helix turn length
    private Vector3 spoolAxis;               // helix axis
    private Vector3 spoolRadial0;            // radial reference direction
    private Vector3 bottomDir;               // radial direction toward the lowest point
    private int minWrappedPoints => keepFirstPointAttached ? 1 : 0; // the last segment is kept attached

    // ------------- Unity -------------
    void Awake()
    {
        lineRenderer = GetComponent<LineRenderer>();
        if (lineRenderer) lineRenderer.useWorldSpace = true;

        if (autoFindRigidbodies)
        {
            if (!startRB && startPointTransform) startRB = startPointTransform.GetComponentInParent<Rigidbody>();
            if (!endRB && endPointTransform) endRB = endPointTransform.GetComponentInParent<Rigidbody>();
        }

        ValidateAndMaybeClampRopeLength();
        InitializeSpoolGeometry();
        InitializeRope();
        RecomputeMaterialAndMass();
        AllocateSolverBuffers();
        DrawRope();
    }

    private IEnumerator Start()
    {
        // It waits for other scripts to have positioned the anchors
        yield return null;
        yield return new WaitForFixedUpdate();

        if (!useInspectorRopeLength)
        {
            if (startPointTransform && endPointTransform)
            {
                float d = Vector3.Distance(startPointTransform.position, endPointTransform.position);
                ropeLengthMeters = Mathf.Max(1e-5f, d);
                segmentLength = ropeLengthMeters / Mathf.Max(1, segmentCount);
                RecomputeMaterialAndMass();
                InitializeSpoolGeometry(); // if segmentLength changes
                InitializeWrappedCountFromExcess(); // It recomputes with the new L
                DrawRope();
            }
        }
```

```
        else
        {
            ValidateAndMaybeClampRopeLength();
        }

        if (setEndRbCenterOfMass && endRB)
        {
            endRB.centerOfMass = endRbCenterOfMassLocalOffset;
            endRB.WakeUp();
        }
        if (setStartRbCenterOfMass && startRB)
        {
            startRB.centerOfMass = startRbCenterOfMassLocalOffset;
            startRB.WakeUp();
        }

        // CSV init
        InitializeCsv();

        // It activates the deployment if there are wrapped points
    beyond the minimum
        deploymentActive = (wrappedPointCount > minWrappedPoints);
        deploymentComplete = !deploymentActive;

        yield return new WaitForFixedUpdate();
    }

    void OnValidate()
    {
        segmentCount = Mathf.Max(2, segmentCount);

        if (!lineRenderer) lineRenderer = GetComponent<LineRenderer
    >();

        if (!Application.isPlaying)
        {
            ValidateAndMaybeClampRopeLength();
            InitializeSpoolGeometry();
            InitializeRope();
            RecomputeMaterialAndMass();
            AllocateSolverBuffers();
            DrawRope();
        }
    }

    void FixedUpdate()
    {
        if (ropePoints.Count == 0) return;

        // ------ It updates spool rotation + release  ------
        UpdateSpoolAndRelease();

        // ------ SUBSTEPS ------
        int S = Mathf.Max(1, solverSubsteps);
        float dtSub = Time.fixedDeltaTime / S;
```

```
        float dt2Sub = dtSub * dtSub;
        lastDt2Sub = dt2Sub;

        if (warmstart && lambdas != null)
        {
            float k = Mathf.Clamp01(lambdaDecay);
            for (int i = 0; i < lambdas.Length; i++) lambdas[i] *= k;
        }
        else
        {
            System.Array.Clear(lambdas, 0, lambdas.Length);
        }

        for (int s = 0; s < S; s++)
        {
            // It imposes the positions of the wrapped points before integration
            ApplySpoolLockPositions();

            Integrate(dtSub);
            SolveTimeStep_Tolerance(dt2Sub);

            // It re-sets the positions of the wrapped points after solving to ensure adherence
            ApplySpoolLockPositions();
        }

        ComputeTensionFromLambdas(lastDt2Sub);

        if (applyBackReaction) ApplyBackReactionForces();

        DrawRope();

        // ---- CSV sample write ----
        CsvWriteSample();

        if (enableSanityCheck) SanityCheck();
    }

    private void OnDestroy()
    {
        CloseCsv();
    }

    private void OnApplicationQuit()
    {
        CloseCsv();
    }

    // ------------- CSV helpers -------------
    private void InitializeCsv()
    {
        if (!enableCsvLogging) return;
        if (string.IsNullOrEmpty(csvDirectory))
        {
```

```
            csvDirectory = Application.persistentDataPath;
            Debug.LogWarning($"[Rope CSV] Directory non impostata. Uso: {csvDirectory}");
        }

        try
        {
            Directory.CreateDirectory(csvDirectory);
            csvFullPath = Path.Combine(csvDirectory, $"{csvBaseFileName}.csv");


            if (File.Exists(csvFullPath))
                File.Delete(csvFullPath);

            csvWriter = new StreamWriter(csvFullPath, append: false, Encoding.UTF8);
            csvWriter.AutoFlush = true;


            csvWriter.WriteLine("Time,PosX,PosY,PosZ,Tension");

            Debug.Log($"[Rope CSV] (overwrite) Logging to: {csvFullPath}");
        }
        catch (System.Exception ex)
        {
            Debug.LogError($"[Rope CSV] CSV initialization error: {ex.Message}");
            enableCsvLogging = false;
        }
    }


    private void CsvWriteSample()
    {
        if (!enableCsvLogging || csvWriter == null) return;

        Vector3 pos;
        if (endPointTransform) pos = endPointTransform.position;
        else if (ropePoints.Count > 0) pos = ropePoints[segmentCount].currentPosition;
        else pos = Vector3.zero;

        float t = Time.time;

        string line = string.Format(CultureInfo.InvariantCulture,
            "{0:F6},{1:F6},{2:F6},{3:F6},{4:F6}",
            t, pos.x, pos.y, pos.z, Tension);

        try
        {
            csvWriter.WriteLine(line);
        }
        catch (System.Exception ex)
```

```
        {
            Debug.LogError($"[Rope CSV] CSV write error: {ex.Message}");
        }
    }

    private void CloseCsv()
    {
        try
        {
            if (csvWriter != null)
            {
                csvWriter.Flush();
                csvWriter.Dispose();
                csvWriter = null;
                // Debug.Log($"[Rope CSV] File cloifd: {csvFullPath}");
            }
        }
        catch (System.Exception ex)
        {
            Debug.LogError($"[Rope CSV] CSV close error: {ex.Message}");
        }
    }

    // ------------- Spool helpers -------------
    private void InitializeSpoolGeometry()
    /* It is responsible for constructing the helical geometry that governs the tether's adhesion to and
    unwinding from the spool. It also identifies the physical reference directions necessary for the
    deployment logic */
    {
        // Helix axis
        if (spoolTransform)
        {
            spoolAxis = spoolTransform.up.normalized;

            // radial direction
            spoolRadial0 = spoolTransform.right.normalized;

            // helix turn length
            float circ = 2f * Mathf.PI * Mathf.Max(1e-5f, spoolRadius);
            helixTurnLength = Mathf.Sqrt(circ * circ + helixPitch * helixPitch);

            // radial direction toward the lowest point
            bottomDir = Vector3.ProjectOnPlane(Vector3.down, spoolAxis);
            if (bottomDir.sqrMagnitude < 1e-8f)
            {
                bottomDir = -spoolRadial0;
            }
```

```
            else bottomDir.Normalize();
        }
        else
        {
            spoolAxis = Vector3.up;
            spoolRadial0 = Vector3.right;
            bottomDir = Vector3.down;
            helixTurnLength = Mathf.Max(1e-5f, 2f * Mathf.PI * Mathf.
Max(1e-5f, spoolRadius)); // pitch ignored if there is no spool
        }

        InitializeWrappedCountFromExcess();
    }

    private void InitializeWrappedCountFromExcess()
    // It determines the number of tether segments that are wound on
the spool at the beginning of the simulation
    {
        // It computes the excess: L_rope - distanza_anchor -
extraSlackMargin
        float d = 0f;
        if (startPointTransform && endPointTransform)
            d = Vector3.Distance(startPointTransform.position,
endPointTransform.position);

        float excess = Mathf.Max(0f, ropeLengthMeters - d - Mathf.Max
(0f, extraSlackMargin));
        int segExcess = Mathf.Clamp(Mathf.FloorToInt(excess / Mathf.
Max(1e-5f, segmentLength)), 0, segmentCount - 1);

        // wrappedPointCount = how many points are wound on the spool
        wrappedPointCount = segExcess;
        if (wrappedPointCount < minWrappedPoints) wrappedPointCount =
 minWrappedPoints;
    }

    private Vector3 HelixPosition(float sAlongHelix, float
extraAngleOffsetRad)
    // It computes the spatial position of tether points constrained
to lie on the surface of the spool
    {
        // Helical parametrization
        float theta = (sAlongHelix / Mathf.Max(1e-5f, helixTurnLength
)) * (2f * Mathf.PI) + extraAngleOffsetRad;

        Vector3 axis = spoolAxis;
        Vector3 u = spoolRadial0;
        Vector3 v = Vector3.Cross(axis, u).normalized;

        float R = Mathf.Max(1e-5f, spoolRadius);
        float z = (sAlongHelix / Mathf.Max(1e-5f, helixTurnLength)) *
 helixPitch;

        Vector3 radial = (Mathf.Cos(theta) * u + Mathf.Sin(theta) * v
) * R;
```

```
        Vector3 axial = axis * z;

        return (spoolTransform ? spoolTransform.position : (startPointTransform ? startPointTransform.position : transform.position)) + radial + axial;
    }

    private Vector3 HelixRadialDir(float sAlongHelix, float extraAngleOffsetRad)
    /* It computes the direction associated with a tether point at a given arclength along the helix.
    This direction vector is critical for the release logic */
    {
        float theta = (sAlongHelix / Mathf.Max(1e-5f, helixTurnLength)) * (2f * Mathf.PI) + extraAngleOffsetRad;
        Vector3 axis = spoolAxis;
        Vector3 u = spoolRadial0;
        Vector3 v = Vector3.Cross(axis, u).normalized;
        Vector3 radial = (Mathf.Cos(theta) * u + Mathf.Sin(theta) * v).normalized;
        return radial;
    }

    private void ApplySpoolLockPositions()
    /* It enforces the kinematic adhesion of the wound portion of the tether to the spool surface at the
    beginning and end of each physics substep */
    {
        // Points [0 .. wrappedPointCount-1] are adhering and locked on the spool
        for (int i = 0; i < wrappedPointCount; i++)
        {
            float s = i * segmentLength;
            Vector3 pos = HelixPosition(s, angleOffset);
            RopePoint p = ropePoints[i];
            p.currentPosition = pos;
            p.previousPosition = pos;
            p.isLocked = true;
            ropePoints[i] = p;
            invMass[i] = 0f; // locked
        }

        // It ensures that point 0 always remains attached
        if (keepFirstPointAttached && wrappedPointCount <= 0 && ropePoints.Count > 0)
        {
            RopePoint p0 = ropePoints[0];
            p0.currentPosition = HelixPosition(0f, angleOffset);
            p0.previousPosition = p0.currentPosition;
            p0.isLocked = true;
            ropePoints[0] = p0;
            invMass[0] = 0f;
        }
    }

```

```
    private void UpdateSpoolAndRelease()
    /* It advances the spool kinematics and decides whether the outermost wound point should be
    released during the current physics step */
    {
        if (!spoolTransform) return;

        // 1) Kinematic rotation
        if (driveSpoolRotation && deploymentActive && !deploymentComplete)
        {
            float dAngle = spoolAngularSpeed * Time.fixedDeltaTime;
            spoolTransform.Rotate(spoolAxis, Mathf.Rad2Deg * dAngle, Space.World);
            angleOffset += dAngle; // It updates offset for the points on the spiral
        }

        // 2) If there are wrapped points beyond the minimum, checks the outermost point to be released
        if (deploymentActive && wrappedPointCount > minWrappedPoints)
        {
            int iOuter = wrappedPointCount - 1;

            float s = iOuter * segmentLength;
            Vector3 radial = HelixRadialDir(s, angleOffset);
            float dot = Vector3.Dot(radial, bottomDir);

            float cosWindow = Mathf.Cos(releaseWindowDeg * Mathf.Deg2Rad);
            if (dot >= cosWindow)
            {
                // Release
                RopePoint p = ropePoints[iOuter];
                p.isLocked = false;
                p.previousPosition = p.currentPosition;
                ropePoints[iOuter] = p;

                float massPerPoint = linearDensity * segmentLength;
                invMass[iOuter] = (massPerPoint > 0f ? 1f / massPerPoint : 0f);

                wrappedPointCount--;

                if (wrappedPointCount <= minWrappedPoints)
                {
                    FinishDeployment();
                }
            }
        }
    }

    private void FinishDeployment()
    /* When the number of wound points falls to the minimum value, the code marks deployment as inactive
```

```
    and complete */
    {
        deploymentActive = false;
        deploymentComplete = true;

        // It stops the spool
        if (spoolRB)
        {
            spoolRB.angularVelocity = Vector3.zero;
            spoolRB.Sleep();
        }
        if (driveSpoolRotation)
        {
            spoolAngularSpeed = 0f;
        }
    }

    // ------------- Core -------------
    private void InitializeRope()
    {
        ropePoints.Clear();

        segmentLength = Mathf.Max(1e-5f, ropeLengthMeters / Mathf.Max
(1, segmentCount));

        Vector3 a = startPointTransform ? startPointTransform.
position : transform.position;
        Vector3 b = endPointTransform ? endPointTransform.position :
transform.position + Vector3.right * ropeLengthMeters;

        InitializeSpoolGeometry();

        for (int i = 0; i <= segmentCount; i++)
        {
            RopePoint rp = new RopePoint();

            if (i < wrappedPointCount)
            {
                float s = i * segmentLength;
                Vector3 pos = HelixPosition(s, angleOffset);
                rp.currentPosition = pos;
                rp.previousPosition = pos;
                rp.isLocked = true;
            }
            else
            {
                Vector3 exitPos;
                if (wrappedPointCount > 0)
                    exitPos = HelixPosition(wrappedPointCount *
segmentLength, angleOffset);
                else
                    exitPos = a;

                float t = (i - wrappedPointCount) / Mathf.Max(1f, (
segmentCount - wrappedPointCount));
```

```
                Vector3 target = snapStraightAtStart ? Vector3.Lerp(
    exitPos, b, t) : Vector3.LerpUnclamped(exitPos, b, t);
                rp.currentPosition = target;
                rp.previousPosition = target;
                rp.isLocked = (i == segmentCount); // the last one is
     anchor B
            }

            if (i == 0)
                rp.isLocked = true;

            ropePoints.Add(rp);
        }
    }

    private void RecomputeMaterialAndMass()
    /* It computes effective cross-section area, linear mass density,
     Young modulus * area (EA),
     and XPBD compliance. */
    {
        float r = Mathf.Max(1e-6f, ropeDiameter * 0.5f);
        crossSectionArea = Mathf.PI * r * r;
     // A = pi r^2
        linearDensity = crossSectionArea * Mathf.Max(1f, density); //
     mu = A*rho

        float EA = Mathf.Max(youngModulus * crossSectionArea, 1e-9f);
        float cBase = segmentLength / EA;

        int S = Mathf.Max(1, solverSubsteps);
        float cEff = scaleComplianceWithSubsteps ? (cBase / (S * S))
    : cBase;

        compliance = new float[segmentCount];
        for (int i = 0; i < segmentCount; i++)
            compliance[i] = cEff;

        invMass = new float[segmentCount + 1];
        float massPerPoint = linearDensity * segmentLength;
        for (int i = 0; i <= segmentCount; i++)
        {
            bool endLocked = (i == 0 || i == segmentCount);
            bool onSpool = (i < wrappedPointCount) || (
    keepFirstPointAttached && i == 0);
            invMass[i] = (endLocked || onSpool) ? 0f : (massPerPoint
    > 0f ? 1f / massPerPoint : 0f);
        }
    }

    private void AllocateSolverBuffers()
    // It allocates and initializes solver buffers: Lagrange
    multipliers (lambdas) and per-segment tensions.
    {
        lambdas = new float[segmentCount];
        SegmentTensions = new float[segmentCount];
```

```
    }

    private void Integrate(float dt)
    // Verlet integration step for free points (non-anchored, non-wound).
    {
        Vector3 gravityStep = new Vector3(0f, gravity * dt * dt, 0f);

        // pin start/end anchors
        if (startPointTransform)
        {
            var p = ropePoints[0];
            p.currentPosition = HelixPosition(0f, angleOffset);
            p.previousPosition = p.currentPosition;
            ropePoints[0] = p;
        }
        if (endPointTransform)
        {
            var p = ropePoints[segmentCount];
            Vector3 pos = endPointTransform.position;
            p.currentPosition = pos;
            p.previousPosition = pos;
            ropePoints[segmentCount] = p;
        }

        // Verlet for free points
        for (int i = 1; i < segmentCount; i++)
        {
            if (ropePoints[i].isLocked) continue;

            RopePoint p = ropePoints[i];
            Vector3 vel = (p.currentPosition - p.previousPosition) * damping;
            Vector3 old = p.currentPosition;
            p.currentPosition += vel + gravityStep;
            p.previousPosition = old;
            ropePoints[i] = p;
        }
    }

    private void SolveTimeStep_Tolerance(float dt2)
    // It iterates constraint solving and collisions until the error falls below tolerance or iterations cap.
    {
        float maxErr = float.PositiveInfinity;
        int it = 0;

        while (it < iterations && maxErr > constraintTolerance)
        {
            if (handleCollisions) HandleSegmentCollisions();
            ApplyXPBDConstraints(dt2);

            maxErr = ComputeMaxConstraintError();
            it++;
        }
```

```

        // It re-pins the ends to prevent any drift
        if (startPointTransform)
        {
            var p = ropePoints[0];
            p.currentPosition = HelixPosition(0f, angleOffset);
            p.previousPosition = p.currentPosition;
            ropePoints[0] = p;
        }
        if (endPointTransform)
        {
            var p = ropePoints[segmentCount];
            p.currentPosition = endPointTransform.position;
            p.previousPosition = p.currentPosition;
            ropePoints[segmentCount] = p;
        }
    }

    private void ApplyXPBDConstraints(float dt2)
    // It applies XPBD distance constraints to enforce segment rest
lengths and accumulate lambdas.
    {
        float omega = Mathf.Clamp(sor, 1f, 1.99f);

        for (int i = 0; i < segmentCount; i++)
        {
            Vector3 xi = ropePoints[i].currentPosition;
            Vector3 xj = ropePoints[i + 1].currentPosition;

            Vector3 d = xi - xj;
            float dist = d.magnitude;
            if (dist < 1e-9f) continue;

            float C = dist - segmentLength; // bilateral
            Vector3 n = d / dist;

            float w1 = invMass[i];
            float w2 = invMass[i + 1];
            float wsum = w1 + w2;
            if (wsum <= 0f) continue;

            float alpha = compliance[i] / dt2; // alpha = c_eff /
dt_sub^2
            float dlambda = (-C - alpha * lambdas[i]) / (wsum + alpha
);
            dlambda *= omega; // SOR

            lambdas[i] += dlambda;

            if (w1 > 0f) { var p = ropePoints[i]; p.currentPosition
+= w1 * dlambda * n; ropePoints[i] = p; }
            if (w2 > 0f) { var p = ropePoints[i + 1]; p.
currentPosition += -w2 * dlambda * n; ropePoints[i + 1] = p; }
        }
    }
```

```
    private float ComputeMaxConstraintError()
    // It computes the maximum constraint violation (distance error) across all segments.
    {
        float maxErr = 0f;
        for (int i = 0; i < segmentCount; i++)
        {
            float dist = Vector3.Distance(ropePoints[i].currentPosition, ropePoints[i + 1].currentPosition);
            float err = Mathf.Abs(dist - segmentLength);
            if (err > maxErr) maxErr = err;
        }
        return maxErr;
    }


    private void HandleSegmentCollisions()
    // It handles segment collisions by sampling subpoints and projecting them outside colliders.
    {
        float ropeRadius = Mathf.Max(1e-5f, ropeDiameter * 0.5f);

        for (int i = 0; i < segmentCount; i++)
        {
            if (ropePoints[i].isLocked && ropePoints[i + 1].isLocked) continue;

            Vector3 A = ropePoints[i].currentPosition;
            Vector3 B = ropePoints[i + 1].currentPosition;
            Vector3 mid = (A + B) * 0.5f;

            float r = ropeRadius * 0.6f;

            // Sphere overlap test; if penetration, it pushes out along normal
            var cols = Physics.OverlapSphere(mid, r, collisionLayers, QueryTriggerInteraction.Ignore);
            foreach (var col in cols)
            {
                if (spoolTransform && col.transform.IsChildOf(spoolTransform)) continue;

                Vector3 closest = col.ClosestPoint(mid);
                Vector3 dir = mid - closest;
                float sq = dir.sqrMagnitude;
                if (sq < 1e-12f) dir = mid - col.bounds.center;
                if (dir.sqrMagnitude < 1e-12f) continue;

                float dist = dir.magnitude;
                float pen = r - dist;
                if (pen <= 0f) continue;

                Vector3 corr = (dir / dist) * pen;

```

```
                if (!ropePoints[i].isLocked)
                {
                    var p = ropePoints[i];
                    p.currentPosition += corr * 0.5f;
                    ropePoints[i] = p;
                }
                if (!ropePoints[i + 1].isLocked)
                {
                    var p = ropePoints[i + 1];
                    p.currentPosition += corr * 0.5f;
                    ropePoints[i + 1] = p;
                }
            }
        }
    }

    private void ComputeTensionFromLambdas(float dt2)
    // It recovers segment tensions from Lagrange multipliers after
XPBD.
    {
        float tMax = 0f;

        for (int i = 0; i < segmentCount; i++)
        {
            float Ti = (-lambdas[i]) / dt2; // Tension
            if (Ti < 0f) Ti = 0f;
            SegmentTensions[i] = Ti;
            if (Ti > tMax) tMax = Ti;
        }

        Tension = tMax;
    }


    private void ApplyBackReactionForces()
    // It applies equal and opposite reaction forces to the endpoints
    (rigidbodies).
    {
        float dt2 = lastDt2Sub > 0f ? lastDt2Sub : (Time.
fixedDeltaTime * Time.fixedDeltaTime);

        // START side (i=0)
        if (startRB && !startRB.isKinematic && segmentCount >= 1)
        {
            Vector3 p0 = ropePoints[0].currentPosition;
            Vector3 p1 = ropePoints[1].currentPosition;
            Vector3 n0 = (p1 - p0).sqrMagnitude > 1e-12f ? (p1 - p0).
normalized : Vector3.zero;

            float T0 = Mathf.Max(0f, -lambdas[0] / dt2); // [N]
            if (T0 > 0f && n0 != Vector3.zero)
                startRB.AddForceAtPosition(+T0 * n0, p0, ForceMode.
Force); // Toward tether
        }

```

```
        // END side (i=ifgmentCount-1)
        if (endRB && !endRB.isKinematic && segmentCount >= 1)
        {
            Vector3 pn_1 = ropePoints[segmentCount - 1].currentPosition;
            Vector3 pn = ropePoints[segmentCount].currentPosition;
            Vector3 nN = (pn - pn_1).sqrMagnitude > 1e-12f ? (pn - pn_1).normalized : Vector3.zero;

            float TN = Mathf.Max(0f, -lambdas[segmentCount - 1] / dt2); // [N]
            if (TN > 0f && nN != Vector3.zero)
                endRB.AddForceAtPosition(-TN * nN, pn, ForceMode.Force); // Toward tether
        }
    }

    private void DrawRope()
    // It draws the rope polyline through the current point positions using a LineRenderer.
    {
        if (!lineRenderer) return;

        int n = ropePoints.Count;
        if (lineRenderer.positionCount != n)
            lineRenderer.positionCount = n;

        for (int i = 0; i < n; i++)
            lineRenderer.SetPosition(i, ropePoints[i].currentPosition);
    }

    // --------- Sanity ----------
    private void ValidateAndMaybeClampRopeLength()
    // It ensures ropeLengthMeters is not shorter than the anchor distance; optionally clamp.
    {
        if (!(startPointTransform && endPointTransform)) return;

        float d = Vector3.Distance(startPointTransform.position, endPointTransform.position);

        if (clampRopeLengthToAnchors && ropeLengthMeters < d)
        {
            Debug.LogWarning($"[Rope] ropeLengthMeters ({ropeLengthMeters:F4} m) < anchor distance ({d:F4} m). Clamp at distance.");
            ropeLengthMeters = d;
        }
        else if (ropeLengthMeters < d)
        {
            Debug.LogWarning($"[Rope] ropeLengthMeters ({ropeLengthMeters:F4} m) < anchor distance ({d:F4} m). Unfeasible state: increase the length or bring the anchors closer.");
        }
```

```
    }

    private float MaxSagFromChord()
    // It estimates the maximum sag based on chord length and slack (approximate catenary).
    {
        if (!(startPointTransform && endPointTransform)) return 0f;

        Vector3 a = startPointTransform.position;
        Vector3 b = endPointTransform.position;

        float maxSag = 0f;
        for (int i = 0; i <= segmentCount; i++)
        {
            float t = i / (float)segmentCount;
            Vector3 onLine = Vector3.Lerp(a, b, t);
            float sag = (onLine.y - ropePoints[i].currentPosition.y);
            if (sag > maxSag) maxSag = sag;
        }
        return maxSag;
    }

    private void SanityCheck()
    // It performs consistency checks (arc length vs. target, sag limits, constraint error) and log summaries.
    {
        _frameCount++;
        if (_frameCount % Mathf.Max(1, logEveryNFrames) != 0) return;
        if (!(startPointTransform && endPointTransform)) return;

        float d = Vector3.Distance(startPointTransform.position, endPointTransform.position);
        float L0 = segmentLength * segmentCount;
        float arc = CurrentArcLength();
        float deltaL = arc - ropeLengthMeters;
        float slack = Mathf.Max(0f, ropeLengthMeters - d);
        float maxSag = MaxSagFromChord();
        float sagLim = maxSagFraction * ropeLengthMeters;

        float arcErrAbs = Mathf.Abs(deltaL);
        float arcErrRel = Mathf.Abs(deltaL) / Mathf.Max(ropeLengthMeters, 1e-6f);
        if (arcErrAbs > arcTolAbs && arcErrRel > arcTolRel)
            Debug.LogWarning($"[Rope] Arc!=L: arc={arc:F6} m, L={ropeLengthMeters:F6} m, Delta={deltaL:+0.000000;-0.000000} m, tolAbs={arcTolAbs:E2}, tolRel={arcTolRel:P2}");

        if (Mathf.Abs(L0 - ropeLengthMeters) > 1e-6f)
            Debug.LogWarning($"[Rope] Inconsistent: segmentLength*count={L0:F6} m different from ropeLengthMeters={ropeLengthMeters:F6} m");

        bool sagTooLarge = (maxSag > sagLim) || (slack > 0f && maxSag > sagVsSlackFactor * slack);
        if (sagTooLarge)
```

```
            Debug.LogWarning($"[Rope] Excessive sag: maxSag={maxSag:F4} m, sagLim={sagLim:F4} m, slack={slack:F4} m. MaxTension={Tension:F2} N");

        if (warnOnHighConstraintError)
        {
            float maxErr = ComputeMaxConstraintError();
            if (maxErr > warnConstraintFactor * constraintTolerance)
                Debug.LogWarning($"[Rope] Constraints error: max|dist-L0|={maxErr:E3} m (> {warnConstraintFactor}xtol). Increase iterations/substeps or rigidity (E).");
        }

        // Compact informational log
        Debug.Log($"[Rope] d={d:F4}  L={ropeLengthMeters:F4}  arc={arc:F4}  slack={slack:F4}  sag={maxSag:F4}  Tmax={Tension:F2}N wrapped={wrappedPointCount}");
    }

    public float CurrentArcLength()
    // It computes the current rope arc length by summing segment distances.
    {
        float L = 0f;
        for (int i = 0; i < segmentCount; i++)
            L += Vector3.Distance(ropePoints[i].currentPosition, ropePoints[i + 1].currentPosition);
        return L;
    }
}
```

Appendix G

PIDTensionController_Deployment. - Deployment

```
/*
 * Project    : A digital twin of an air-bearing platform for tethered satellite systems:
                from tether deployment to post-deployment control
 * File       : PIDTensionController.cs - Deployment case
 * Author     : Edoardo De Blasi
 * Supervisors: Prof. Paolo Maggiore, Dr. Giuseppe Governale, Prof. Stephanie Lizy-Destrez
 * Date       : September 2025
 * Notes      : Developed in Unity using C# scripting.
 * License    : This code is intended for academic and research purposes only.
*/
using UnityEngine;
using System;
using System.Reflection;


/// <summary>
/// PID controller that regulates tether tension in real time and can stop the spool at a deployed-length threshold.
/// </summary>
public class PIDTensionController : MonoBehaviour
{
    [Header("Scene Objects")]
    public XPBDTetherWithSpool tetherRope;
    public Rigidbody floater;
    public Transform bench;
    public Transform tetherAnchor;

    [Header("PID Control Settings")]
    public float desiredTension = 50f;
    public float Kp = 1.0f;
    public float Ki = 0.1f;
    public float Kd = 0.2f;

    [Header("Scaling")]
```

```
    public bool applyScaling = true;
    public enum ScalingScenario { HCW_Emulation, Tether_Deployed }
    public ScalingScenario scenario = ScalingScenario.Tether_Deployed
;
    [Tooltip("Override lambdaL e lambdat. Lascia 0 per auto-fill
dallo scenario.")]
    public float lambdaL = 0f;  // length scale (real/table)
    public float lambdaT = 0f;  // time scale (real/table)

    [Header("Simulation Settings")]
    public float maxAngleDeg = 3.0f;

    [Header("PID Safeguards & Filters")]
    [Tooltip("Clamp integrale per evitare windup (tensione x s)")]
    public float integralClamp = 1000f;
    [Tooltip("Low-pass filter (Hz) on the measured tension")]
    public float tensionLPFCutoffHz = 3f;
    [Tooltip("Optional clamp on the commanded acceleration (m/s^2)")]
    public float outputAccClamp = 5f;

    [Header("Spool stop by deployed length")]
    [Tooltip("Stop the spool when the deployed length (free segments
* L0) exceeds this threshold (m).")]
    public bool stopSpoolByLength = true;
    public float deployedLengthStopMeters = 1.0f;
    public bool stopOnce = true;

    [Header("Actuator Simulation")]
    public float actuatorSmoothTime = 0.2f; // time to reach the
target (s)

    [Header("Debug / Probe")]
    public bool verboseLogs = true;
    public int logEveryNFrames = 10;
    public bool dryRunThreshold = false;

    // --- Debug visibile da Inspector ---
    [SerializeField] private float dbgSegLen = -1f;
    [SerializeField] private int   dbgSegmentCount = -1;
    [SerializeField] private int   dbgWrappedNow = -1;
    [SerializeField] private int   dbgFreeSegments = -1;
    [SerializeField] private float dbgDeployedLen = 0f;
    [SerializeField] private bool  dbgThresholdReached = false;

    // Actuator state
    private float currentRoll, currentPitch;
    private float rollVelocity, pitchVelocity;

    // PID state
    private float filteredTension = 0f;
    private float integral = 0f;
    private float previousError = 0f;

    private bool stopIssued = false;
    private int frameCounter = 0;
```

```
    private bool thresholdLoggedOnce = false;

    private PropertyInfo piWrappedCount, piSegmentLength,
piSegmentCount;
    private FieldInfo fiWrappedCount, fiSegmentLength, fiSegmentCount
;
    private MethodInfo miForceStopSpoolPublic;

    void Start()
    {
        if (applyScaling)
        {
            if (lambdaL <= 0f || lambdaT <= 0f)
            {
                if (scenario == ScalingScenario.HCW_Emulation) {
lambdaL = 700f; lambdaT = 500f; } // Scaling factors with only HCW
 equations, no tether
                else { lambdaL = 50f;  lambdaT = 20f;  } // In case
of tethered floater
            }
        }

        if (tetherRope != null) filteredTension = tetherRope.Tension;

        CacheRopeIntrospection();
    }

    void FixedUpdate()
    {
        ComputeControl();
    }

    private void ComputeControl()
    {
        if (tetherRope == null || floater == null || bench == null ||
 tetherAnchor == null) return;

        float dt = Time.fixedDeltaTime;
        if (dt <= 0f) return;

        // === PID ===
        float cutoff = Mathf.Max(0f, tensionLPFCutoffHz);
        if (cutoff > 0f)
        {
            float alpha = Mathf.Exp(-2f * Mathf.PI * cutoff * dt);
            filteredTension += (1f - alpha) * (tetherRope.Tension -
filteredTension);
        }
        else
        {
            filteredTension = tetherRope.Tension;
        }

        // Errors computation
        float error = desiredTension - filteredTension;
```

```
        integral += error * dt;
        if (integralClamp > 0f) integral = Mathf.Clamp(integral, -integralClamp, integralClamp);
        float derivative = (error - previousError) / dt;
        previousError = error;

        // Control acceleration computation
        float outputAcceleration = Kp * error + Ki * integral + Kd * derivative;

        Vector3 dir = floater.position - tetherAnchor.position;
        dir.y = 0f;
        dir = (dir.sqrMagnitude > 1e-9f) ? dir.normalized : Vector3.zero;

        Vector3 controlAcceleration = dir * outputAcceleration;

        // Scaling for emulation (a_table = (lambdat^2/lambdaL) * a_real)
        if (applyScaling)
        {
            float scaleAcc = (lambdaT * lambdaT) / Mathf.Max(1e-6f, lambdaL);
            controlAcceleration *= scaleAcc;
        }

        // Optional clamp on commanded acceleration to avoid extreme saturation
        if (outputAccClamp > 0f)
        {
            float mag = controlAcceleration.magnitude;
            if (mag > outputAccClamp) controlAcceleration *= (outputAccClamp / mag);
        }

        float g = 9.81f;
        float maxAngleRad = Mathf.Deg2Rad * Mathf.Max(0.1f, maxAngleDeg);

        // Conversion of the control acceleration into angles (in radians)
        // small angle: a = g * theta
        float targetRollRad  = Mathf.Clamp( controlAcceleration.z / g, -maxAngleRad, maxAngleRad);
        float targetPitchRad = Mathf.Clamp(-controlAcceleration.x / g, -maxAngleRad, maxAngleRad);

        // Basic anti-windup
        {
            float desiredRollRad  =  controlAcceleration.z / g;
            float desiredPitchRad = -controlAcceleration.x / g;
            float satErr = (desiredRollRad - targetRollRad) + (desiredPitchRad - targetPitchRad);
            // small coefficient to avoid destabilizing (tuning):
            integral -= 0.1f * satErr * dt;
```

```
        }

        // Actuator smoothing working in degrees to use SmoothDampAngle
        float targetRollDeg  = targetRollRad  * Mathf.Rad2Deg;
        float targetPitchDeg = targetPitchRad * Mathf.Rad2Deg;

        float currentRollDeg  = currentRoll  * Mathf.Rad2Deg;
        float currentPitchDeg = currentPitch * Mathf.Rad2Deg;

        // It applies SmoothDamp to the angles (works in degrees)
        currentRoll  = Mathf.SmoothDampAngle(currentRollDeg, targetRollDeg,  ref rollVelocity,  actuatorSmoothTime) * Mathf.Deg2Rad;
        currentPitch = Mathf.SmoothDampAngle(currentPitchDeg, targetPitchDeg, ref pitchVelocity, actuatorSmoothTime) * Mathf.Deg2Rad;

        // It applies the smoothed rotation
        bench.rotation = Quaternion.Euler(currentRoll * Mathf.Rad2Deg, 0f, currentPitch * Mathf.Rad2Deg);

        // deployedLen = freeSegments * ifgmentLength
        frameCounter++;

        float segLen        = GetSegmentLength(tetherRope);
        int   segmentCount  = GetSegmentCount(tetherRope);
        int   wrappedNow    = GetWrappedPointCount(tetherRope);

        int freeSegments = -1;
        float deployedLen = -1f;

        if (segLen > 0f && segmentCount >= 0 && wrappedNow >= 0)
        {
            freeSegments = Mathf.Clamp(segmentCount - wrappedNow, 0, segmentCount);
            deployedLen  = freeSegments * segLen;

            bool reached = deployedLen >= deployedLengthStopMeters;

            // debug -> Inspector
            dbgSegLen = segLen;
            dbgSegmentCount = segmentCount;
            dbgWrappedNow = wrappedNow;
            dbgFreeSegments = freeSegments;
            dbgDeployedLen = deployedLen;
            dbgThresholdReached = reached;

            // periodic log
            if (verboseLogs && (logEveryNFrames <= 0 || (frameCounter % logEveryNFrames) == 0))
            {
```

```
                Debug.Log($"[PID] Deployed probe: segLen={segLen:F4} segCount={segmentCount} wrapped={wrappedNow} free={freeSegments} deployedLen={deployedLen:F3} thr={deployedLengthStopMeters:F3}");
            }

            // one-shot log when threshold is exceeded
            if (reached && !thresholdLoggedOnce)
            {
                thresholdLoggedOnce = true;
                Debug.LogWarning($"[PID] *** THRESHOLD REACHED *** deployedLen={deployedLen:F3} m >= {deployedLengthStopMeters:F3} m => {(dryRunThreshold ? "DRY-RUN" : "STOP")}");
            }

            // stop
            if (stopSpoolByLength && !stopIssued && reached && !dryRunThreshold)
            {
                StopSpool(tetherRope);
                if (stopOnce) stopIssued = true;
            }
        }
        else
        {
            if (verboseLogs && (logEveryNFrames <= 0 || (frameCounter % logEveryNFrames) == 0))
            {
                Debug.Log($"[PID] Deployed probe NOT EVALUABLE: segLen={segLen}, segCount={segmentCount}, wrapped={wrappedNow}");
            }
        }
    }

    // === Stops the spool ===
    private void StopSpool(XPBDTetherWithSpool rope)
    {
        if (miForceStopSpoolPublic != null)
        {
            try
            {
                miForceStopSpoolPublic.Invoke(rope, null);
                if (verboseLogs) Debug.Log("[PID] ForceStopSpool() invoked (public).");
                return;
            }
            catch (Exception ex)
            {
                Debug.LogWarning($"[PID] ForceStopSpool() invocation failed: {ex.Message}");
            }
        }

        // fallback hard
        try
```

```
        {
            rope.spoolAngularSpeed = 0f;
            rope.driveSpoolRotation = false;
            if (rope.spoolRB != null)
            {
                rope.spoolRB.angularVelocity = Vector3.zero;
                rope.spoolRB.Sleep();
            }
            Debug.LogWarning("[PID] Hard stop fallback applied.");
        }
        catch (Exception ex)
        {
            Debug.LogWarning($"[PID] Hard stop fallback error: {ex.
Message}");
        }
    }


    private void CacheRopeIntrospection()
    {
        if (tetherRope == null) return;
        var t = typeof(XPBDTetherWithSpool);
        const BindingFlags PUB = BindingFlags.Instance | BindingFlags
.Public;
        const BindingFlags NP  = BindingFlags.Instance | BindingFlags
.NonPublic;


        piWrappedCount   = t.GetProperty("WrappedPointCount",  PUB);
        piSegmentLength  = t.GetProperty("SegmentLengthMeters", PUB);
        piSegmentCount   = t.GetProperty("SegmentCount",        PUB);

        // private fields for fallback
        fiWrappedCount   = t.GetField("wrappedPointCount", NP);
        fiSegmentLength  = t.GetField("segmentLength",     NP);
        fiSegmentCount   = t.GetField("segmentCount",      NP);

        // preferred public method
        miForceStopSpoolPublic = t.GetMethod("ForceStopSpool", PUB);
    }

    private int GetWrappedPointCount(XPBDTetherWithSpool rope)
    {
        try
        {
            if (piWrappedCount != null) return (int)piWrappedCount.
GetValue(rope, null);
            if (fiWrappedCount != null) return (int)fiWrappedCount.
GetValue(rope);
        }
        catch { }
        return -1;
    }

    private float GetSegmentLength(XPBDTetherWithSpool rope)
```

```
    {
        try
        {
            if (piSegmentLength != null) return Convert.ToSingle(piSegmentLength.GetValue(rope, null));
            if (fiSegmentLength != null) return (float)fiSegmentLength.GetValue(rope);
        }
        catch { }
        return -1f;
    }

    private int GetSegmentCount(XPBDTetherWithSpool rope)
    {
        try
        {
            if (piSegmentCount != null) return (int)piSegmentCount.GetValue(rope, null);
            if (fiSegmentCount != null) return (int)fiSegmentCount.GetValue(rope);
        }
        catch { }
        return -1;
    }
}
```