POLITECNICO DI TORINO

Master Degree course in Aerospace Engineering

Master Degree Thesis

Digital Twin for new-generation lunar pressurized rovers

Supervisors
Prof. Giuseppe Palaia
Prof. Karim Abu Salem
Prof. Enrico Zappino

Candidate
Damiano Prezioso - 324279

Academic Year 2025-2026

Abstract

This thesis aims to review the role and evolution of Digital Twins and to develop a step-by-step method to build a Digital Twin for a space application. In particular, this thesis has developed the Digital Twin for a docking system specially developed for lunar surface exploration, within Italian space research program Space It Up! The work thus offers a method to move from the definition of the Digital Twin scope to the validation and testing phases. The review of Digital Twins starts from their conception to the rapid spread of the last years, also considering advanced implementations, with a focus on the space sector. The review of the state-of-the-art highlighted the absence of peer consensus on the method and definition. Taking its cue from this, the thesis proposes itself as a guiding tool for the development of a custom-made Digital Twin. The design process starts by defining scope, boundary conditions, and requirements. Through a preliminary functional analysis, the architecture of the Digital Twin has been defined, distinguishing between the physical model which describes the actual phenomenon of the docking system, and the Digital Twin, which connects the digital model to the real body. Alongside the actual development and the integration with the real system, verification, calibration and validation stages are covered. The digital twin of a 1:10 scaled prototype docking system has been developed, supervising and controlling its operations. The Digital Twin starts from an Adams multi-body model and is based on MATLAB-Simulink; software-hardware communication relies on Arduino. A test campaign is performed and described to verify the proper functioning of the Digital Twin.

Abstract - Versione italiana

La presente tesi si propone di esaminare il ruolo e l'evoluzione dei Digital Twin, 'gemelli digitali', e di sviluppare un metodo passo-passo per la progettazione di un Digital Twin destinato ad applicazioni spaziali. In particolare, nel lavoro di tesi è stato sviluppato il Digital Twin di un sistema di docking progettato appositamente per l'esplorazione della superficie lunare, nell'ambito del programma italiano di ricerca spaziale Space It Up! Il lavoro vuole offrire quindi una metododologia per passare dalla definizione dell'ambito operativo del Digital Twin fino alla fase di validazione e collaudo. La rassegna preliminare sulla tecnologia Digital Twins parte dalla prima formulazione, fino alla recente e rapida diffusione, considerando anche applicazioni avanzate con un focus sul settore spaziale. Tale rassegna dello stato dell'arte ha evidenziato la mancanza di consenso tra gli esperti circa le metodologie e una definizione univoca. Prendendo spunto da ciò, la tesi si propone come una guida di alto livello per lo sviluppo di un Digital Twin su misura. Il processo progettuale inizia con la definizione del campo di lavoro, delle condizioni al contorno e dei requisiti. Attraverso una preliminare analisi funzionale, viene definita l'architettura del gemello digitale, distinguendo tra il modello matematico-virtuale che descrive il fenomeno fisico e la struttura software del Digital Twin, che collega il modello virtuale al corpo reale. Parallelamente all'effettivo sviluppo del Digital Twin e alla sua integrazione con il sistema reale, vengono affrontate le fasi di verifica, calibrazione e validazione. È stato quindi sviluppato il gemello digitale di un prototipo in scala 1:10 di un sistema di docking, in grado di supervisionarne e controllarne le operazioni. Lo sviluppo del Digital Twin parte da un modello multi-body, creato con il software Adams; inoltre, esso è basato sull'ambiente MATLAB-Simulink, mentre la comunicazione software-hardware è gestita con Arduino. E' stata infine eseguita e descritta una breve campagna di test per verificarne il corretto funzionamento.

Contents

List of Figures 7

List of Tables 11

1 Acronyms 13

2 Introduction 15

3 State-of-the-art 19
- 3.1 Digital Twin: origins and state-of-the-art . . . 19
  - 3.1.1 The Digital Twin . . . 19
  - 3.1.2 Excursus: Internet of Things . . . 21
  - 3.1.3 Digital Twins in Space . . . 22
  - 3.1.4 Development of a Digital Twin: guidelines from literature . . . 24
- 3.2 Docking and berthing . . . 26
  - 3.2.1 Mating operations . . . 26
  - 3.2.2 Docking characterization . . . 27
  - 3.2.3 Docking architectures . . . 28
- 3.3 Docking for pressurized rovers . . . 33
  - 3.3.1 Pressurized rover interfaces: inputs and outputs . . . 33
  - 3.3.2 Small Pressurized Rover . . . 36
  - 3.3.3 MGAAMA concept . . . 37

4 Requirements 39
- 4.1 The lunar environment . . . 39
  - 4.1.1 Gravity . . . 39
  - 4.1.2 Atmosphere . . . 40
  - 4.1.3 Ionizing radiation . . . 40
  - 4.1.4 Lunar day cycle . . . 40
  - 4.1.5 Temperatures . . . 40
  - 4.1.6 Soil and dust . . . 41
  - 4.1.7 Trafficability . . . 41
  - 4.1.8 Meteoroid bombardment . . . 42
  - 4.1.9 Launch expense . . . 42
- 4.2 The Lunar Pressurized Rover . . . 42

4.2.1 Space It Up programme 42
4.2.2 The Lunar Pressurized Rover 43
4.3 Why the docking system 44
4.4 Previous work 45
4.4.1 Lunar Pressurized Rover docking analysis 45
4.4.2 The docking prototype 47
4.4.3 Minor hardware changes 53
4.5 Digital Twin requirements 53
4.5.1 Layered model of Digital Twins functions 53
4.5.2 The requirements 55

5 Preliminary analysis 57
5.1 Model preliminary analysis 57
5.1.1 Functional analysis 57
5.1.2 Model architecture 60
5.1.3 Study cases 62
5.1.4 Model selection 64
5.2 Digital Twin preliminary analysis 69
5.2.1 Functional analysis 69
5.2.2 Architecture of the Digital Twin 73
5.2.3 Software selection 79

6 Digital Twin development 83
6.1 Review on sensing 83
6.1.1 Sensors: an introduction 83
6.1.2 Virtual sensors 86
6.1.3 Middleware and virtual sensors 89
6.2 Sensors selection 91
6.2.1 Sensors options for the docking prototype 91
6.2.2 Trade-off 94
6.3 Model development and Digital Twin structure 101
6.3.1 Adams model description 101
6.3.2 Real-time solution: the surrogate model 111
6.3.3 Middleware structure 119
6.3.4 Arduino firmware 139
6.4 Simulation instructions 141
6.5 Software verification 142
6.6 Additional algorithms 143

7 Implementation of the Digital Twin 145
7.1 Sensors architecture 145
7.2 Sensors integration 146
7.3 Integration of the Digital Twin 146
7.3.1 Introduction to co-simulations 146
7.3.2 Best-practice for models integration 149

7.3.3 Digital Twin integration 150
7.4 Validation of the model 151
7.4.1 Surrogate model validation 152
7.5 Calibration of the Digital Twin 161

8 Experimental campaign 165
8.1 Tests definition 165
8.1.1 Basic verification test 165
8.1.2 Nominal tests 165
8.1.3 Fault injection tests 166
8.2 Tests results and analysis 167
8.2.1 Failed tests 167
8.2.2 Successful basic tests 167
8.2.3 Limitations: why the Digital Twin is not working 172
8.2.4 Requirements verification 174

9 Integration with the LPR 177

10 Conclusions and further developments 179
10.1 Following developments 180

11 Appendix A: Arduino firmware code 183

Bibliography 197

List of Figures

3.1 Schematic representation of a Digital Twin, depicted as of Grieves original definition. 20
3.2 Suggested general iterative method to develop an environmental model, based on [33] 25
3.3 Berthing scheme, from [36]. 27
3.4 Gemini docking procedure, from [39]. *I* is approach, *II* soft docking, *III* is the final hard-docked configuration. 29
3.5 Soyuz docking gear, in the updated version for astronaut transit, from [40] 30
3.6 Representation of an astronaut removing the docking gear during an apollo mission.[41] Note the three-armed beam-attenuator structure. 30
3.7 Drawings of the two halves, at left the Soviet one, at right the American one. [42] 31
3.8 Technical drawing of the IDSS docking interface, with the inward petals. In the drawing, the umbilicals are hinted, allowing electrical and liquid flow passage. [37] 32
3.9 OECS docking system. [1] 32
3.10 Picture of the Common Berthing Mechanism. Notice the petals, labelled as 7. [43] 33
3.11 Small Pressurized Rover (SPR) during desert tests.[45] Details of the docking flange are shown, with the male angular petals in black. 36
3.12 Pictures of the Active-Active Mating Adapter (AAMA) prototype. The two pictures above (external and internal) are from [46], the CAD model is from [45]. Poor quality of images is from their original source and does not derive from the reproduction process. 37
3.13 MGAAMA CAD model.[45] 38

4.1 Space it Up programme logo.[52] 43
4.2 Sectional schematization of the double torus docking interface.[1] 46
4.3 Pair of active and passive hooks adopted as Hard Docking mechanism. [1] 47
4.4 Visual CAD representation of the LPR and the lunar base, as in the CAD imagined by Binetti. [1] 47
4.5 Formal docking procedure milestones, as defined by [1]. From top to bottom: initial state, initial contact, final contact. 48

4.6 Misalignments introduced by the actuators. At top there is Δ, introduced by *Linear guide 1*. At bottom, it is the γ inserted by the *stepper motor*. [1] 51
4.7 Isometric view of the prototype assembly, from [1]. 52
4.8 Levels of sophistication of a Digital Twin; based on [11]. 54

5.1 Architecture diagram of the model. 61
5.2 Nominal acceleration, speed and position profiles, imposed by Linear Guide 1, as from Binetti's thesis. [1] 63
5.3 Architectural diagram of the Digital Twin. The *Elaboration module* may be considered both inside or outside the DT. 74
5.4 Command and data flow from the prototype to the digital model. To be read from top to bottom. 76
5.5 Commands and data flow from the digital model to the prototype. To be read from bottom to top. 77
5.6 Logical temporal sequence of a single time-step. 78

6.1 Scheme of a closed control loop. [57] 83
6.2 Schematic representation of a virtual sensor, based on [58]. At left, a choice of inputs is shown, including physical sensors on the real asset and a virtual sensors; in the centre is the data fusion function which returns data, as seen on the right as output. 87
6.3 Configuration matrix of the sensor options. 97
6.4 Prioritization matrix with the weight of each figure of merit. Higher score means higher relevance in the trade-off; only outputs of the comparison matrices are shown, with ranking on the right. 99
6.5 Results of the trade-off for the sensors. 100
6.6 Image of the complete Adams model. Notice the reference triad in the low left corner. At left there is the Target, with the support structure; at right, the Chaser (black docking interface is visible). 102
6.7 Rear detail of the female docking interface. Notice the position of springs (two longitudinal and one perpendicular are visible) and the absence of screws, nuts and washers in the visualization. 105
6.8 Docked view of Adams digital model. 108
6.9 Visual representation of an exemplifying Latin Hypercube sampling over a 2D space, with *m=5*. On the x-coordinates a uniform segmentation has been performed, while on the y-axis the sectors show different weights.[71] 114
6.10 Middleware overview in Simulink. Colours have been used to facilitate subsystems recognition. Callback functions (MATLAB code launched before compiling) are not visible in the plot, being written separately, in another window. 120
6.11 Detailed zoom of the middleware overview. The focus is on the first subsystems, activating after initialization with MATLAB functions and after model compiling, but preliminary to the switch to real-time. 121
6.12 Detailed zoom of the real-time subsystem. 121
6.13 Window asking the user run settings. 122

6.14 The figure shows the logic internal to the Enable subsystem, used for sending values to Arduino. 125
6.15 Example of the messaged shown in the Diagnostic Viewer. Here, messages about the initialization process are readable. 126
6.16 Overview of the real-time cycle subsystem (internal view). Blocks have been coloured to facilitate their description. 127
6.17 Simulink commands for starting a new run. 141
6.18 Window asking to enter the time-step. 141
6.19 Window asking to enter the firmware identifier code. 142

7.1 Validation approaches for the surrogate model: direct method and transitive method. 153
7.2 Identity plot for p_{end} polynomial. 154
7.3 Comparison plot for p_{end} polynomial and the original multi-body model. . 155
7.4 Residual plot for p_{end} polynomial. 155
7.5 Identity plot for v_{end} polynomial. 156
7.6 Comparison plot for v_{end} polynomial and the original multi-body model. . 157
7.7 Residual plot for v_{end} polynomial. 157
7.8 Identity plot for $a_{target,RMS}$ polynomial. 158
7.9 Comparison plot for $a_{target,RMS}$ polynomial and the original multi-body model. 159
7.10 Residual plot for $a_{target,RMS}$ polynomial. 159
7.11 Comparison of the theoretical real-time cycle time-step with the actual (average) performed by Simulink. 162

8.1 Nominal test programme. The schedule includes one test for each configuration, but more repetitions of each test are preferred. 166
8.2 Test description of the trials for evaluating the response of the Digital twin to fault injection. 167
8.3 Experimental set-up for testing the Digital Twin in the lab. 168
8.4 Speed values during a simulation are shown. The blue step function represents the objective velocity, that the actuator tries to reach at the and of each step, by translating this value in electrical commands for *Linear Guide 1* (which was actually not able to follow). The red dots represents the output of simulated sensor + filter, at each time instant. 170
8.5 Top figure shows the evolution of target-chaser distance during one simulation, with the speed evolution depicted by red points in figure 8.4. The figure beneath is showing the evolution of angular misalignment γ values, as output of the simulated sensor+ filter; the red dashed line represents the requirement at docking as from Binetti's indications (table 4.4.2). The blue step function represents the commands for the *Stepper motor*. 171

List of Tables

4.1 Requirements from [1]: on top are shown three of the requirements used for the design of the 1:1 scale docking mechanism. On bottom, are listed the requirements of the 1:10 docking prototype, used to define the experimental tests. 49
4.2 Functional requirements of the Digital Twin, with rationales. 55
4.3 Performance requirements of the Digital Twin, with rationales. 56
4.4 Functional requirements of the Digital Twin, with rationales. 56

6.1 Naive miscellany of possible sensors to be integrated with the prototype. . 92
6.2 List of all model parts (created by aggregating actual CAD parts). All parts group more than one component, except when mentioned. 104

Chapter 1

Acronyms

AAMA: Active-Active Mating Adapter
AI: Artificial Intelligence
APAS: Androgynous Peripheral Attachment System
ASI: Agenzia Spaziale Italiana (Italian Space Agency)
CAD: Computer Aided Design
DARPA: Defense Advanced Research Projects Agency
DOE: Design Of Experiments
DT: Digital Twin
DTE: Digital Twin Environment
DTI: Digital Twin Instance
DTP: Digital Twin Prototype
ECLSS: Environmental Control and Life Support System
ECSS: European Cooperation for Space Standardization
ESA: European Space Agency
EVA: Extra Vehicular Activity
FMI: Functional Mock-up Interface
GNC: Guidance, Navigation & Control
GUI: Graphical User Interface
HFL: High Level Architecture
IIoT: Industrial Internet of Things
IMU: Inertial Measurement Unit
IoT: Internet of Things
ISS: International Space Station
LEO: Low Earth Orbit
LHS: Latin Hypercube Sampling
LPR: Lunar Pressurized Rover
LVDT: Linear Variable Differential Transformer
MEMS: Micro-Electro-Mechanical Systems
MGAAMA: Multi-Gravity Active-Active Mating Adapter
NASA: National Aeronautics and Space Administration
NRMSE: Normalized Root Mean Square Error

PLA: Polylactic acid
PGM: Predictive Graphical Model
R&D: Research and Development
RMS: Root Mean Square
RSM: Response Surfaces Method
SI: Système international d'unités
SPR: Small Pressurized Rover
TPU: Thermoplastic polyurethane
TRL: Technology Readiness Level
VV&T: Validation, Verification & Testing

Chapter 2

Introduction

In recent years, with the growth of connectivity and computing power, the concept of *Digital Twin* is gaining ground, as a tool for optimizing and making autonomous Engineering systems. Among the many fields of application that will benefit, space exploration is no exception.
But, before moving to a specific implementation, one could ask: what is a Digital Twin? *Digital Twins* do not have an univocal definition, with authors and companies choosing their own. However, it is possible to identify a Digital Twin as a virtual construct that contains the information to mimic the behaviour of a natural or artificial system, with real-time bidirectional exchange of data between it and its digital counterpart. It can describe at any instant the state of the physical twin and simulate possible configurations, supporting development and decision-making, and can be active part in control and management of the physical system.

It clearly is a wide, and sometimes confused, concept, whose boundaries are not well defined. So, how can Digital Twin been an asset in the development of complex systems for future space exploration?
This exact question emerged in the initial stages of the *Lunar Pressurized Rover (LPR)* preliminary design, an Italian research project , as part of the *Space It Up!* project. The question shifts towards how to exploit the benefits of *Digital Twin* technologies in wheeled vehicles for lunar exploration, with a time horizon no sooner than 2035.
As first brick of this study, this thesis aims at defining boundaries and features of *Digital Twins*, beginning to channel them towards the LPR project.

More in details, the thesis has three purposes. The first is to draw up a review of Digital Twins, with a focus on Space sector applications, defining the fundamental characteristics, trying to give example and to clarify what DTs are and what are not.
Secondly, the thesis proposes as a guideline for the development process of a Digital Twin, merging a typical iterative design approach with the necessary steps for building and testing an effective DT.
As demonstration, the thesis also accompanies the method with an operative example, by describing a first iteration of a simple DT. It represents the first design loop of a DT applied to a simple 1:10 scaled prototype, previously built as part of another master's degree thesis.[1] This was conceived for exploring technologies for terrain docking, necessary for

establishing a walkable passage for astronauts between the pressurized rover and a stable lunar habitat.
These points together allow to begin a methodological discussion on how to leverage Digital Twins in the development of new-generations space exploration programs, also exploiting emerging tool such as artificial intelligence and machine learning.

The thesis is organised into the following sections.
After this *Introduction* (Chapter 2), Chapter 3, *State-of-the-art*, contains the literature review. In particular, a first part of it is about Digital Twins, with the fundamental subsection 3.1.4, with the proposed methodological guidelines for developing a Digital Twin, also employed later in the work. The section also covers *docking technologies*, firstly with the state of the art on spacecraft solutions in micro-gravity, and then with the proposed approaches for manned pressurized rovers.
Chapter 4 passes from the analysis of previous work to the beginning of the project for the docking prototype's Digital Twin. It indeed covers boundary conditions of the work, from a description of the Lunar environment to an overview of *Space It Up!* and *Lunar Pressurized Rover* projects. Moreover, Binetti's work about the docking prototype, object of the DT project, is briefly described. Finally, the chapter ends with the first design step: the definition of the requirements of the Digital Twin.
Chapter 5 contains important methodological steps for the development of the Digital Twin. It concerns two aspects: the model used for simulating the behaviour of the prototype and the software structure of the DT. In particular, the points covered are functional analysis, architectural definition and the selection of the tools to employ.
Chapter 6 is the most substantial. It starts with the review of possible sensors to employ, in order to then move on to the choice of using a simulated sensor for this preliminary iteration of the Digital Twin project. The most important section contains however the description of the Digital Twin, with some references to the development process, in its main parts: the model, built on *Adams* software and converted into a surrogate model; the *middleware*, the controller, based on *Simulink*; the interface with the prototype, *Arduino's firmware*. Also, instructions for running the DT are briefly presented. Finally, there are indications on the verification process and on possible algorithms that could equip the DT with advanced features.
With chapter 7, methodological steps are presented: how to integrate sensors with the Digital Twin; how to integrate all other software and hardware parts, with a short excursus on *co-simulations*; a short validation of the developed simulator, that, as will be seen, does not yield the desired results; indications on the calibration of the Digital Twin, adjusting free parameters for an optimal behaviour.
The last significant chapter, 8, covers the definition and the results of a brief experimental campaign to test the result of the Digital Twin design process. It is anticipated that the DT will not perform satisfactorily, due to numerous inherited hardware issues. However, the architecture is functional and the method successful.
Finally, the concluding sections are: chapter 10.1, about possible refinements of the Digital Twin, in view of possible further project iterations and improvements, with a focus on the issues emerged during the work; chapter 9 covers possible approaches to integrate this work within the main *LPR* project. Conclusions are in chapter 10,

At last, before *Bibliography*, *Appendix A* contains the firmware code for Arduino.

Chapter 3

State-of-the-art

3.1 Digital Twin: origins and state-of-the-art

3.1.1 The Digital Twin

Digital Twin concept was introduced by professor Michael Grieves in 2002 in his Product Lifecycle Management course at Center University of Michigan. He then formalized it during the following years and attributed the now-used name to NASA's Engineer John Vickers [2]. For many years, Grieves has continued advocating the Digital Twin in many possible applications. Supported by the increased potential of computation and information availability, Digital Twins started to appear in industries in the last decade.

NASA itself had a key role in sponsoring the concept, through its 2012 'Modelling, Simulation, Information Technology & Processing' roadmap.[3][4] The following years, Digital Twin concept exited Space industry, getting broader attention and entering industries R&D plans. [5]

In the coming years, promising contributions may arrive from developing technologies: Machine Learning (ML), Artificial Intelligence (AI), Internet of Thing (IoT), Augmented and Mixed Reality (AR and MR). [6] Thus, Digital Twins may assume a key role in Industry 4.0, leveraging efficiency of performances of products throughout the life-cycle .

Being a developing subject, Digital Twins do not have an univocal definition, with authors sparing no effort in writing their own; see [7] and [8] for a survey on some options from literature. However, the concept is not changing: a Digital Twin is a virtual construct containing the information to mirror a natural or artificial system and its behaviour, with real-time bidirectional exchange of data. It can return the state of the physical counterpart, while anticipating possible configurations, supporting development and decision-making, especially during operational phase. With the growth of computational power and interconnectivity, Digital Twins promise reducing the impact on time and costs for all the industrial sector. However, the potential does not end here, as seen in the next paragraphs.

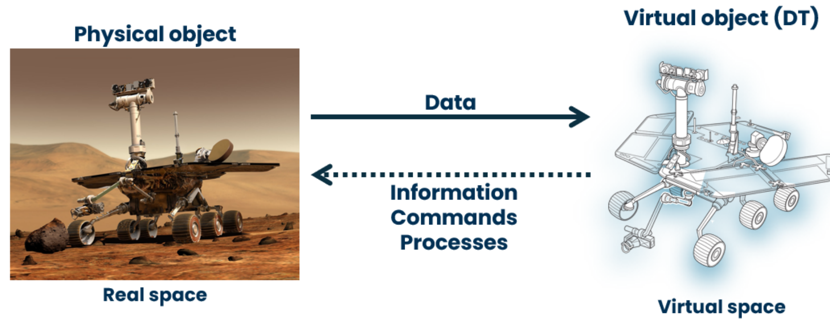

Figure 3.1: Schematic representation of a Digital Twin, depicted as of Grieves original definition.

Grieves himself models the Digital Twin as constituted by three elements [9], as seen in his figure 3.1: the real product, the virtual model, and the connections between them two. The connection with the physical product is necessary to call it a Twin, else being just a simulator.
However, in the beginning of a design phase, it may happen that the virtual model is the only existing part: in order to have a Digital Twin and not just a simulation, it is necessary to prearrange the interaction with the future real body. Grieves defines this early phase concept as the Digital Twin Prototype (DTP) [10]. After the physical version is created, it is no longer a prototype: Grieves also introduces the Digital Twin Instance (DTI), the virtual copy of a physical product that can follow it along its entire life cycle. Thus, Digital Twins are of usefulness during the V-model development, pushing towards a more complete W-model, which includes the design on the DT as well as its employment in the Verification and Validation process (V&V). [11] They may indeed undergo hardware-in-the-loop simulations to have a first feedback from tests, reducing costs of complete hardware testing during the design phases.

For completeness, more complex models of the Digital Twin has been proposed. Instead the three-dimensional model of Grieves, Tao et al. proposed a five-dimensional one: in addition to the physical body, its virtual copy and the interconnection, they add the service dimension and the data dimension. [12] More complex models (e.g. a seven-dimensions DT) has also been proposed but are beyond the scope of this work.

As said, the information stream between the real and the virtual systems is bidirectional. By one side, it is a way to visualize the exact configuration of the physical system. On the other, Digital twins allow to perform complex and high-fidelity simulations, integrating distinct fields of expertise and environmental conditions in a cheap and versatile replica of the system, allowing better operation planning and damage fixing. Thus, it is even possible to predict the behaviour of systems of systems, saving resources and safeguarding the real system from unpredicted harms.

Before establishing the connection with the real body, however, a DT is a digital model. At a general level, three main elements for it are: the entity itself (a rover, a robotic arm, even a human body), created through proper tools; the environment with precise conditions (sometimes called DTE, Digital Twin Environment) [10]; proper sensors

with uncertainties and injected errors. From this, the complexity of the architecture may progressively increase, up to a nearly perfect copy of the real body.

Focusing on the applications of such a technology, all industry may be beneficial. However, fields with rapid integration of Digital Twins are electronic-permeated industries: manufacturing and energy sector, as well as automotive. [6] As well, DT may assist also urban management, resource distribution and traffic flow, also helping public administration.

Enlarging the focus, one of the most complex yet critical systems is planet Earth. As climate change is manipulating its equilibrium, Digital Twins could help in facing its consequences on nature and communities, enhancing the comprehension we have of natural and artificial systems and of their interactions, as well as anticipating catastrophic events.
Space Agencies are indeed carrying out specific projects, combining Earth Observation data, models and Machine Learning to build Digital Twins of specific Earth systems, with forecasting potential. NASA's Earth System Digital Twins (ESDTs) includes flood prediction models, agricultural tools and ecosystems digital twins, as many others. [13] ESA is also timidly moving in the same direction with the Digital Twin Earth program. In the next years, it may represent a solid tool for research and early warning.

3.1.2 Excursus: Internet of Things

Industry was born thanks to mechanization and new emergent energy sources. The Second industrial revolution introduced electricity and mass production enabled by the assembly line. Then, in the 1970s, electronics led to automation and so to Industry 3.0. In current years, technology is heading to an interconnected industry, namely Industry 4.0, with a similar groundbreaking effect.
Industry 4.0 is possible through the Internet of Things, the spreading of objects connected to the Web and able to communicate in a common environment. In particular, when referred to industry, it is named Industrial Internet of Things (IIoT). However, it may be applied to many fields, from health-care to housing (smart homes) and smart cities.[8]

Historically, IoT concept has been introduced in 1990s by Kevin Ashton.[14] It is based on the connection of objects and systems (the *Things*) to the Internet, in order to have a constant monitoring. The concept lies between a cloud Digital Shadow, mirroring the state of the real system, and a Digital Twin, but with lower levels of automation and coverage. As a result, the number of connected devices is constantly increasing: what in the past was exclusive of computers, and then also smartphones, is now possible for fridges, cars, industrial plants, fitness trackers and so on. The possibility to remotely control them and to universally have access to their data opens the path to a magnitude of applications.

A simple consequence of IoT has been however the IPv4 exhaustion case.[15] Each connected device has indeed a recognition code, the IP address. The convention identified IP adreesses with 4 number codes, each number between 0 and 255, for a total of 4 bytes. Such an enumeration allowed less than 4.3 billion available combinations; this system, the *IPv4*, was adopted in the early stages of Internet history, but yet in 2011 resulted not

sufficient to cover all possible devices. The required countermeasure has been to adopt *IPv6*, a new16 bytes address, to support the widespread proliferation of Internet of Things.

3.1.3 Digital Twins in Space

Since NASA's Apollo program, physical twins of spacecrafts has been essential for the success of the missions. They are fundamental for fault resolution of unmanned vehicles, such as wheel slipping or component replacement.[16][17][18] Unfortunately, access to physical twins or analogue environments is expensive and characterized by low availability: another example is Mars Yard, a JPL's utility to simulate Martian and Lunar soil.[19]
Digital Twins may so be a solution: a virtual equivalent of a system such as a rover (or a part of it) is an important resource that can save funds and time, often resulting faster in simulations than the real-time physical counterpart. These advantages represent an attracting opportunity for both government agencies, dealing with critical funding, and for industries, in the context of the privatizing New Space economy. For the aim, Lockheed Martin flanked a Digital Twin to OSIRIS-REx (Origins, Spectral Interpretation, Resource Identification and Security-Regolith Explorer) asteroid sampling mission, focusing on preparing for operations. [20]

As seen, a Digital Twin represents a valid tool to cope with unexpected issues: NASA and Siemens developed a Digital Twin of Curiosity to *solve* problems on the heat management of the RTG (Radioisotope Thermoelectric Generator) during the mission. [21]
Analogously, ESA already flanked Amalia, the twin-on-Earth of planned Rosalind Franklin Martian rover, with a digital version to increase the effectiveness of problem-solving. Some other examples of rover simulators are: NASA's Gazebo and DARTS ROAMS, DLR's URSim and, on a less-specific level, the open source Chrono platform. [22]

One potential of digital twins is to *train* operators in critical and specialized procedures, the same way of simulators (which have been key to staff training for decades): a virtual version of the hardware can be run to evaluate the human-machine interplay and to enhance it. For instance, O'Keefe et al. created a virtual representation of a self-made rover with a mechanical arm, implementing it in a VR environment through Blender and Unity. Two different control groups had to remotely operate the arm. The group with a previous DT training, albeit one, showed better performances, being faster and reporting more ease of operation. [19]
Above all, astronauts may benefit from virtual simulations, with Mixed, Augmented or Virtual Reality: in training phase, extending simulators potential; but also during the mission, with devices that may connect with ground, enhancing social and mental wellness. [6] [23] This idea is however not new, since in the 1990s it was already positively evaluated during Hubble's replacement mission. [24]

Another usage of Digital Twins is as *test beds* for algorithms: it is even possible to build digital twin of environments and integrate them in increasingly more complex models. Since 2003, ESA's software PANGU allows to create scenes of bodies such as the Moon, asteroids and even spacecrafts.[25] Recent progresses in Deep Learning techniques allow to exploit simulators and virtual environments for navigation purposes with high fidelity. With a reverse approach to that of PANGU, Liu et al. assembled a virtual environment the POLAR (Polar Optical Lunar Analog Reconstruction) dataset of Moon's images.[22]

The process was carried out with MATLAB® and the result (surface meshes for creating synthetic images) has been validated with object-detection algorithms.
Another example of testing and training algorithms is offered by. Liu et al.: they built a digital twin model of lunar environment to test an updated algorithm of path planning and obstacle avoidance for an autonomous rover. [26] Exploiting a digital twin consisting of rover, sensors and lunar environment, the authors simulated diverse cases comparing the behaviour of three algorithms, validating the accuracy of the new one.

Digital twins also have the potential to develop *complex simulations* combining multiple interacting systems, such as distinct vehicles. [27] This is more promising when considering systems owned or designed by different groups or agencies in a collaborative international environment. Since future lunar exploration may take advantage of the collaboration of private and public organizations, interconnected virtual simulations may be a useful tool.

Despite its resemblance, a DT is not a true copy of the device. The model used may have increasing levels of detail: in the early stages, simple codes may be sufficiently valid. Among the various languages, Matlab® and the derived Simulink ® environment may be useful tools. This choice was made by Pinello et al. to assemble the vital systems of a rover, used to individuate the effects of hardware failures. [21] A virtual model of a generic rover was created on the base of Perseverance and Rosalind Franklin, composed by systems and sub-systems, each with a specified model and with inputs and outputs. With their work, the authors suggest how Digital Twins may provide a solid base to produce data to develop and train machine learning models for damage detection, such as a HUMS (Health and Usage Monitoring System), helping since the design phase to devise countermeasures.
More complex models may include an interactive 3D version of the body, even with VR application and human interactions. The spread of virtual twins in the last years has been possible thanks to multiplication of digital tools and powerful software.
Here three example are reported, with no aim of exhaustiveness. *Simscape*^{TM} is a Simulink® extension to model and simulate many multidomain physical systems; it is available since 2008. [28] *Project Chrono*, a multi-physics library for simulations of vehicles motion and their interactions with fluids and soil, was developed since 1998 but became open-source only in 2013. [29] *Omniverse* is a real-time collaborative platform for 3D graphics released in 2022 by Nvidia.[30] It is an example of how video-games world has had a role in the development of technologies today crucial for VR and Digital Twins. [31]

Digital Twins for Space applications generally consist of four areas [6]:

- *Mathematical models*, representing the physics of the system and its environment with implementation of laws and valid approximations. Often, a 3D version of the model is built, generally as a CAD model, and associated to the multi-physics laws.

- *Probabilistic estimate algorithms*, such as Kalman filter and particle filter, that can assess the state of the system from real-time data and can predict its future behaviour.

- *Diagnosis algorithms*, used to detect potential damages and even identify possible countermeasures.

- *Optimisation algorithms*, to maximize the efficiency of resources, leveraging the performance. For the sake, parametric models are generally adopted.

The need of a diagnostic algorithm is, specifically, a crucial point for future manned missions outside LEO, where communication delays, occultations and uncrewed operations may represent an hazardous element for which on-ground Mission Control Centres will not be sufficient. Moreover, hardware repairing is not always possible, and early-detecting may save the mission. Enhancing the autonomy of vehicles is thus a vital design requirement for the next-generation of astronautic programmes. For this reason, hierarchical and modular design approaches are cherished; NASA itself promotes the Modular Autonomous System Technology (MAST) as a basis of future spacecraft software design, starting right from Artemis programme.
A last sample of how Digital Twins may aid the development of a space system, Gratius et al. applied a predictive approach to a specific part of the ECLSS (Environmental Control and Life Support System): the temperature management subsystem.[32] A Predictive Graphical Model (PGM), which combine statistics and graphs to generate predictions, has been implemented on a terrestrial testbed (three adjacent offices at Carnegie Mellon University, USA); similar methods had been already exploited for other scopes of application. The authors show how this approach may forecast the evolution of important variable, as temperature or chemical concentrations, preventing exceeding thresholds and individuating possible failures. So, they suggest this approach may increase the safety of deep space habitats, endangered by occultations and communication latency with on-ground control centres

To conclude, future orbital and lunar missions will benefit from a virtual copy, facilitating management, fault identification and operation planning. This measure will help emancipating from Mission Control on Earth, weakened by latency (e.g. 1.3 seconds for Moon-Earth round-trip), data flow limitations and possible occultations. This is even essential for Martian missions, for which the latency rises to many minutes. In particular, being responsible of human life, manned missions would require safety technologies for the most critical systems (such as the ECLSS), being the most favoured scenery for Digital Twins' benefits, in particular for faults prevention.

3.1.4 Development of a Digital Twin: guidelines from literature

To conclude, here it is presented a generic scheme to follow to build a Digital Twin. It has been adapted from the one used by Gratius et al. and integrated with the work of Jakeman et al. "Ten iterative steps in development and evaluation of environmental models" (schematized in figure 3.2). [32] [33].

- Define the purpose of the work.
- Choose the subsystems to be included in the virtual simulation.
- Define the scope of the subsystems and the entire system, by individuating boundary conditions of the rover, variables, outputs.
- Perform a functional analysis to identify requirements and functions, with a conceptualisation of the system; a review of the scope may be beneficial.

- If the physical object is not yet defined, an architecture definition of it has to be delineated.
- Select the models and the software, with adequate integration in case of multi-physics models; motivations should be explicated.
- Design the Digital Twin, verifying it runs correctly; it should be the least complex (at equivalent results) and the most flexible (by using parametric tools). There should be already taken into consideration to include interfaces (both input and output) with the physical asset and a memory to store data.
- If the system needs calibration and training, collect external data and use them. It should be better to assess the uncertainty degrees of the model.
- Validate the model and DT with different data and through peer review; validation is required both for the single models (preferable to be done previously) both for the entire DT after the integration.
- When the real system is ready, twin it to the digital model for real-time communication and integration; this action may require various steps, including careful calibration.

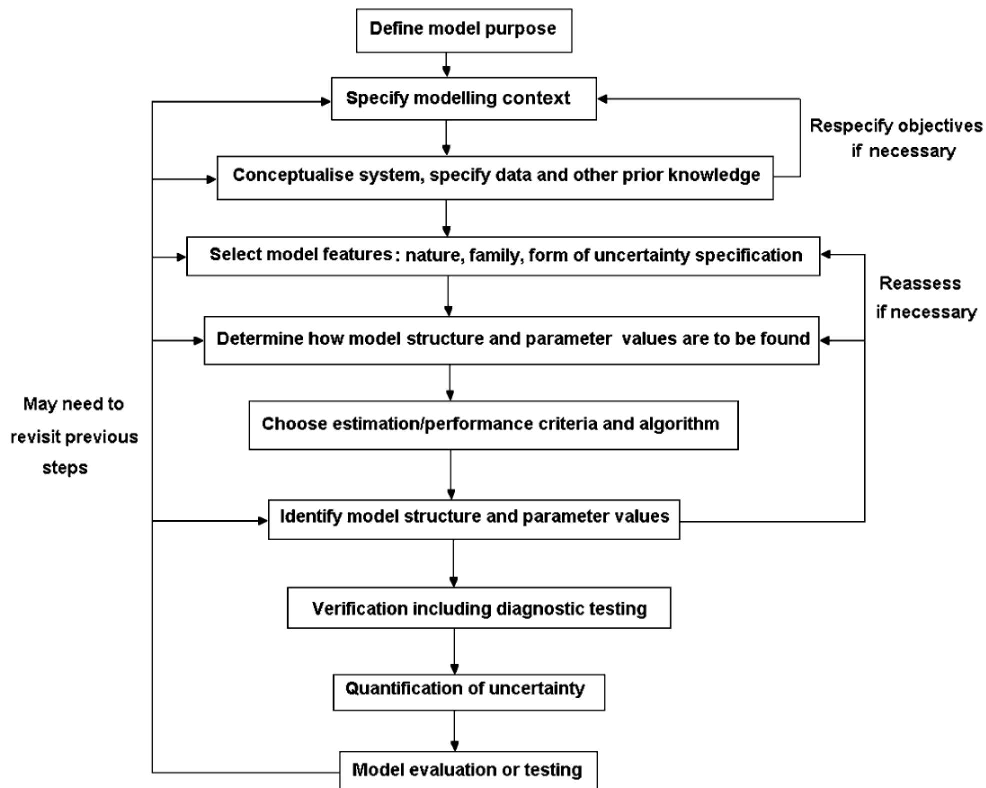

Figure 3.2: Suggested general iterative method to develop an environmental model, based on [33]

In case of negative outcome at any point, it is necessary to return to previous steps, in an iterative process, until the result is satisfying. Moreover,the whole process should be subjected to continuous reporting to produce proper documentation. This point of the work is crucial, aiming to provide an high-level guiding algorithm to create a Digital Twin, hoping to be useful for future works. This thesis follows this outline. However, the composition may slightly vary from a strict adherence due to variability of the architecture. The beginning of each chapter will guide in the placement of the various tasks inside this overall picture. Furthermore, consider that this work represents the first iteration of this guideline, and that the result is far from being a definitive and ready-to-use Digital Twin.

The communication between the physical and the virtual bodies should be managed without unplanned losses, as well as the storage of data in a designated depository. As one of many authors, George suggests the use of a Middleware with this exactly tasks, also representing an interface for some external actors, such as personnel or another software. [34]

3.2 Docking and berthing

3.2.1 Mating operations

Since the first space missions, rendezvous and mating operations appeared as necessary steps for complex architectures and more remarkable results, both technological and scientific. If rendezvous is already complex in itself, mating is the most difficult but enabling result.
The first *rendezvous* has been accomplished in 1965 by NASA astronaut Wally Schirra, by orbiting with Gemini 7 for more than 20 minutes at around 30 centimetres from Gemini 6.[35] Thus, rendezvous means getting close to a target body, artificial but also natural, and keeping the relative distance constant and relative velocities null for some time and in a controlled way.
A more complex manoeuvrer is *mating*, in which the two bodies arrive at stable contact, forming a unique body with a common centre of mass. [1] The two bodies are generally distinguished for their role in the guidance operations:

- The *target*, which is passively mated and has, as only role, to keep its kinematic conditions;
- The *chaser*, which actively conducts the operations.

Mating is intended for artificial spacecrafts and often allows the+ flow of people and goods between the two bodies, creating a unique space. Mating may however have however also other purposes, from debris removal, with just an exchange of energy, to machinery fixing, such as Shuttle's Hubble repair. Neil Armstrong with Gemini 8 had the first mating in spaceflight history, just the year after Gemini 7. Since then, these operations are ordinary and in many conditions, even requiring a remarkable engineering effort.
It is important to discern between two types of mating:

- *Docking*, in which the chaser approaches the target thanks to GNC system control, and arrives directly at contact with the target. It is done by progressively zeroing

relative velocities, both linear and angular, up to touching; the small residual speed is dissipated in the contact. It is assimilable to a rendezvous ending with null relative distance of, at least, a pair of points (one on the target, one on the chaser).

- *Berthing* firstly requires the rendezvous of chaser and target, ending with a given pose between the two bodies. The contact is induced by a manipulator, assembled on one of the two bodies, which grabs the other vehicle and pulls it towards itself, as in fig.3.3. An example of manipulator is the CanadArm (both the one on the Shuttle and the one on the ISS).

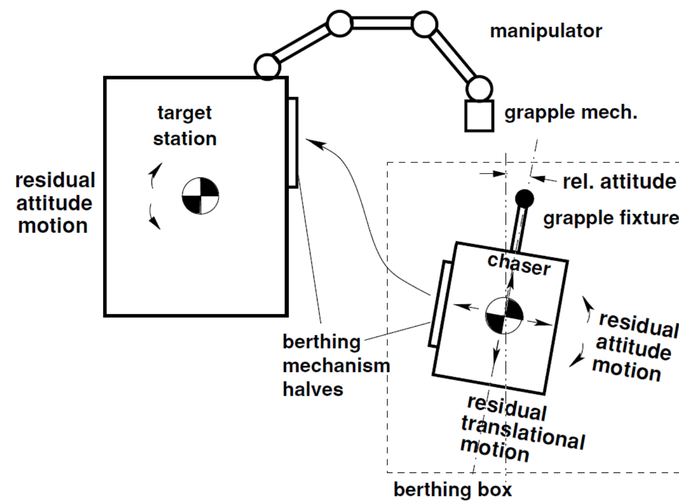

Figure 3.3: Berthing scheme, from [36].

3.2.2 Docking characterization

Focusing on docking, there are two main possible solutions [1]:

- *Central docking system*, in which the chaser has a specific *probe* (or *rod*, or *pin*). This male part is designed to fit in a cavity of the target, called *drogue*. This female portion, with the shape of a conical frustum,guides the pin towards its centre. When the rod reaches its apex, is is said to be a *soft docking*, namely the first controlled contact between the two bodies. This approach is used in a simpler version for in-flight plane refuelling.
  However, this mechanism is just a guide for the second portion of the process, in which the pin is slowly retracted and the chaser progressively aligns to the target with all the final docking surface.
  Eventually, the *hard docking* (or capture) is obtained with the definitive surface contact between the two bodies with null relative velocities and with the sealing through a robust mechanism, such as bolts or hooks.

- *Peripheral docking system*, in which the docking equipment is placed in the peripheral region of the docking regions (in both the bodies). This solution is an evolution of the previous, developed to overtake its main disadvantage: the peripheral solution allows to keep the docking gear out of the docking surface. This allows it to act easily as passage for astronauts after the docking.
  Indeed, the central probe-and-drogue unit (after the mating) hinders a simple tunnel between the two modules, requiring to be removed after the docking to allow the transit. For this reason, peripheral solutions are preferable for manned missions, while central are favoured for many unmanned dockings, in light of their simplicity.

A further evolution is *androgynous docking systems*. This concept, which does not replace the two categories above, while being a possible subcategory, implies an architecture in which both the halves may act passive or active. More precisely, the docking interface is capable of mating to an identically configured interface. [37] It gives a slightly different meaning to *target* and *chaser* notions, which are linked to the GNC behaviour; the important distinction, from a docking perspective, is which interface is active and which is passive from the actuators and brakes perspective. It means indeed that, in case of certain faults, the spacecrafts may switch their roles, the passive one becoming active in docking activities: this property increases flexibility and safety.
The first example of androgynous docking has been adopted in the *Apollo-Soyuz Test Project (ASTP)*, a joint experiment of NASA and Soviet space program, which saw an Apollo and a Soyuz capsule docking. In addition to the strong political meaning, during Cold War season, this project made docking more versatile, opening the door to collaborative missions.
After this 1975 milestone, this system evolved in the *Androgynous Peripheral Attachment System (APAS)*, which has become the most common type of docking system for decades.[1] Currently, its last evolution is the *International Docking System Standard (IDSS)*, a standard docking interface adopted by the main space agencies, each developing their own compatible docking version. The only remarkable exception is China National Space Administration (CNSA), which uses their latest variant of the APAS: *APAS-2010*.

3.2.3 Docking architectures

Here is a synthetic review of the historical most relevant docking solutions. Mohtar's work [38] has been used as reference: see it for a more detailed analysis.

Gemini docking system

The first ever, the Gemini-Agena mechanism combined a male probe (on the Gemini side) with a female cup interface, the drogue, on the non-manned Agena spacecraft. This double frustum (a truncated cone) structure is actually peripheral, but considerable almost central because not large and thus occupying around all the docking surface. Relative velocities, both longitudinal and lateral, were dampened by shock absorbers on the target spacecraft, under its drogue. The precise alignment was achieved with an indexing bar on the male part, which, sliding in a V-shaped guide on the target, allowed the docking (fig.3.4). This single gear causes to have only a possible attitude configuration for docking,

to align the bar and its guide. Finally, a motor, for pulling the two spacecraft together, and latches closed the Hard Docking procedure.

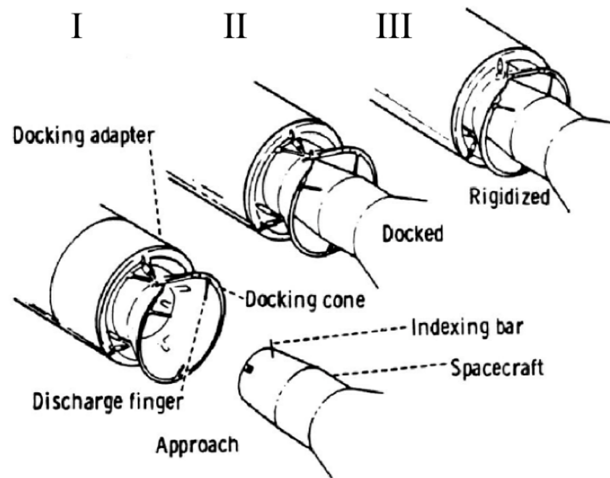

Figure 3.4: Gemini docking procedure, from [39]. *I* is approach, *II* soft docking, *III* is the final hard-docked configuration.

Soyuz docking system

Soviet's Soyuz docking system is based on a central probe-and-drogue architecture and is still in use today, after sixty years from the original development. The probe-and-drogue mechanism directly exploits the probe to align the two halves. Shock attenuation was achieved by the introduction of screw mechanisms such as an electromechanical brake (EMB) and of a frictional brake. Since the original version did not allow the transit of astronauts, an updated one was created, as seen in fig.3.5.

Apollo docking system

Also the Apollo missions saw a probe-and-drogue mechanism, with a central piston, three 120°-disposed centring beams and three piston pitch bungees for shock attenuation. The system was not fully reusable, since dry nitrogen was used for probe retraction and was stored for not more than four retractions (back-ups included). To allow the passage of astronauts, this central mechanism had to be manually removed and stored, as in figure 3.6.

ASTP: the Apollo-Soyuz docking system

By the combined effort of NASA and the Soviet space program, the conciliatory project of Apollo-Soyuz Test Program yielded the first peripheral and androgynous docking system, as already mentioned in section 3.2.2. The docking surface and the latches were indeed adopted commonly by the two spacecrafts. However, technically, under the two halves

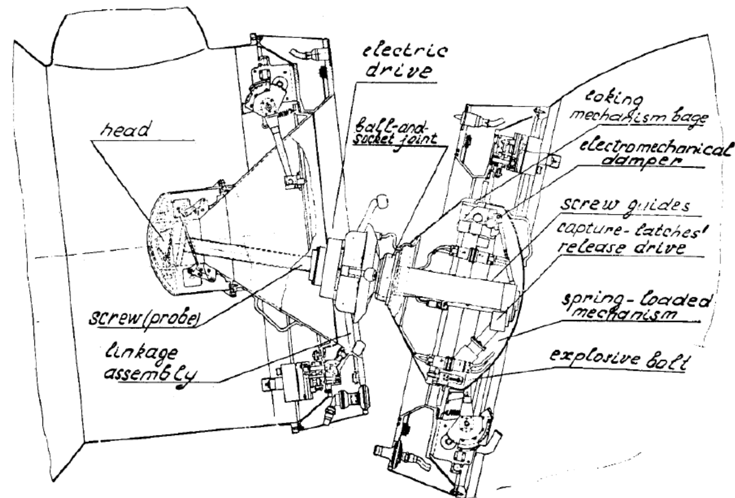

Figure 3.5: Soyuz docking gear, in the updated version for astronaut transit, from [40]

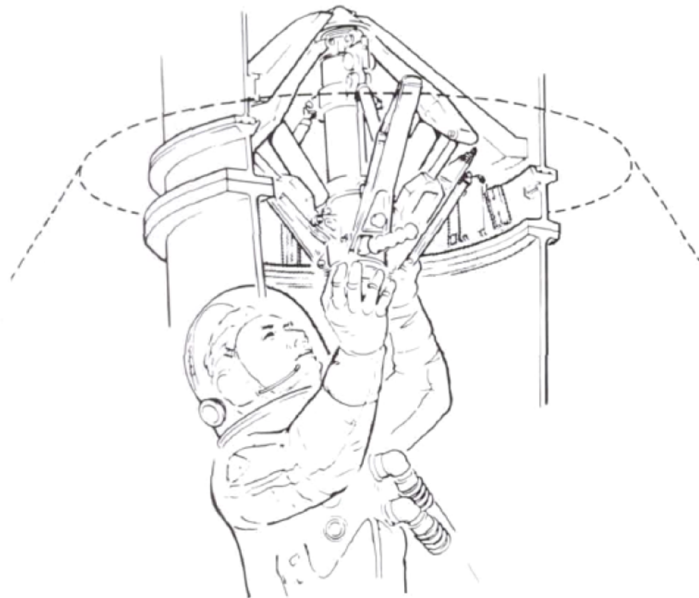

Figure 3.6: Representation of an astronaut removing the docking gear during an apollo mission.[41] Note the three-armed beam-attenuator structure.

there was different machinery behind the common interface, especially for the actuation procedure. The common surface firstly showed the divergent guiding petals as in figure 3.7

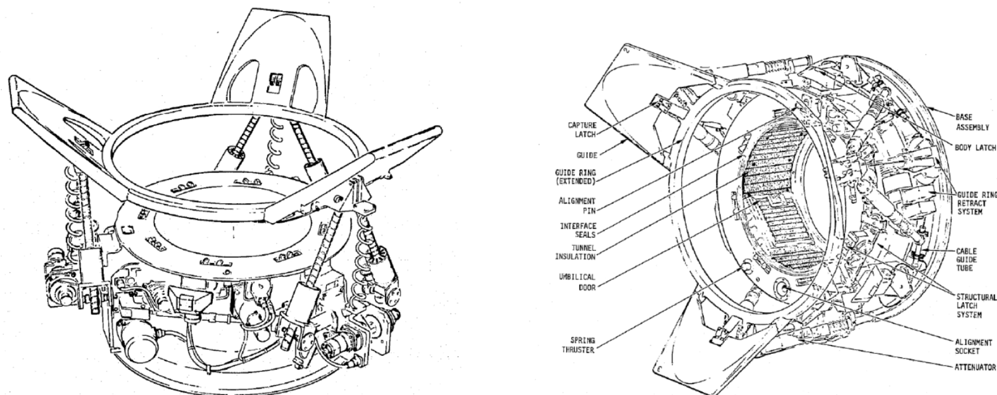

Figure 3.7: Drawings of the two halves, at left the Soviet one, at right the American one. [42]

Androgynous Peripheral Attachment System (APAS)

The APAS (Androgynous Peripheral Attachment System) is a category of docking interfaces developed on the basis of the ASTP docking system. A version was the APAS-89 developed for the unfortunate Soviet spaceship Buran; a new element is the convergence of the petals, instead of the divergent ones seen in figure 3.7.
A 1995 version was instead developed for the ISS (as well as Shuttle and Mir) and intensively used. It is important to underline that the International Space Station (ISS), despite being in orbit since 1998, is generally taken as state-of-the-art reference for any manned mission design. The reason is its more-than-twenty-years service, with a long chronology of integrations, faults and upgrades, as well for the big amount of collected data. A newest version has been developed in 2010 for the Chinese space program, as already mentioned.
The APAS mechanism exploits a Stewart-Gough platform on the active side to have a good precision on six degrees of freedom and to align the two sides.

International Docking System Standard (IDSS)

The last evolution of the APAS is the International Docking System Standard (IDSS), a standard adopted by many Space agencies for future programs, as in figure 3.8. The American version is the NASA Docking System (NDS), while the European one is the International Berthing and Docking Mechanism (IBDM). Again, the important feature is the interface, and not the methods to ensure requirements achievement.

Claw systems: the OECS test

DARPA tested the Orbital Express Capture System (OECS) in the docking manoeuvrers of two satellites, Astro and NEXTSat. The halves are non-androgynous, one being a system of three fingers and a common actuator. The passive side is a three-tip wedge at which the active claws may clasp. Docking through claws may be an easy solution

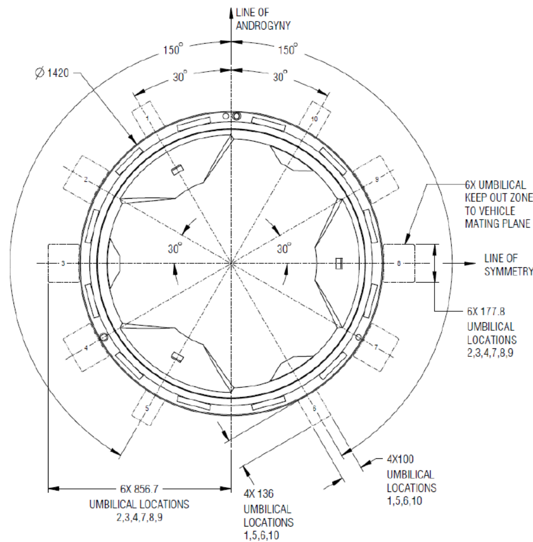

Figure 3.8: Technical drawing of the IDSS docking interface, with the inward petals. In the drawing, the umbilicals are hinted, allowing electrical and liquid flow passage. [37]

for unmanned satellites, even of small dimensions, while this solution is not feasible for manned architectures requiring the union of inhabited modules. A promising application is instead for deorbiting of previous launched satellites, designed without a proper dismission plan.

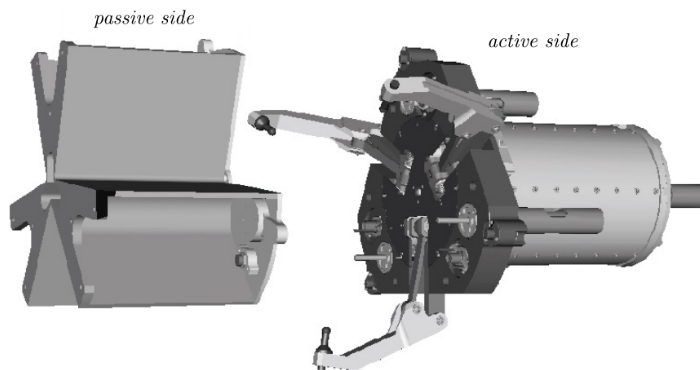

Figure 3.9: OECS docking system. [1]

Common Berthing Mechanism (CBM)

This non-androgynous system allows the ISS berthing manoeuvres with cargo ships. It has a large transit surface for loading and unloading operations, with small convergent petals. An image is shown in figure 3.10. Since berthing is not the focus of this topic, not much is said about.

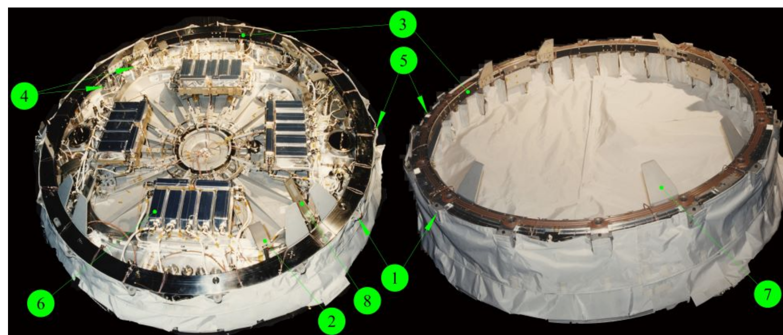

Figure 3.10: Picture of the Common Berthing Mechanism. Notice the petals, labelled as 7. [43]

Nozzle docking mechanism

Proposed innovative solutions involve mission extension of exhaust satellites. It is obtained by docking, as passive side, the nozzle (or nozzles) of the satellite to push. The active side may both refuel the passive one if allowed by design, both act as an attached propulsion system. This approach may also be a solution for dismission, as seen for the claw systems.

Electromagnetic docking mechanism

Another proposed soft docking architecture exploits electromagnetic field's self-aligning capabilities. This type of solution may be effective for small mass satellites, such as CubeSats. It is currently only a proposed configuration.

3.3 Docking for pressurized rovers

If, by one side, docking and berthing in micro-gravity are well-known instances, with space stations built and ISS long story, things changes when considering a planet surface. In particular, for manned vehicles, the state-of-the-art is still more in its early steps, with few developments and no effective missions completed. Indeed, the only six crewed missions on another planet date back to the Apollo program. It represented one of the most notable results of humankind, but still based on quite simple mission architectures and dated technologies.
For many years on, early projects of pressurized rovers for lunar exploration have been started (and many of them also concluded, as just prove-of-concept). However, more extensive studies of the docking system are lacking. Here, a brief, non-comprehensive, review is conducted.

3.3.1 Pressurized rover interfaces: inputs and outputs

Starting with an architectural point of view, it is clear pressurized rovers need controlled interfaces with the unpressurised outside. Here it is a list of the possible interfaces with the exterior or with other artificial environments.

Access from another habitat

As the main object of this work, one of the main interfaces is the access for the astronauts. It may be a direct bridge from another environment, in particular from a stable habitat. It must be noted that, unlike in microgravity, the crew cannot float, and a floor or a vertical step must be granted.
For a horizontal bridge, the passage section must be bigger than in microgravity, to allow astronauts to walk. A thinner passage area is sufficient for vertical docking (from the ceiling of the rover to the floor of the habitat), which instead requires a ladder.

The passage may be *pressurized*, allowing astronauts to walk without spacesuits and easily carrying objects. A careful design concern is the pressurization atmospheric values (typically fully defined as composition, pressure and temperature) to adopt, both for the habitat and the rover. If they are kept at different pressures and/or compositions, this passage has to be kept separated from both the environments; this requires time for locking one side and bringing the tunnel's atmospheric conditions to the ones on the other side. Only after that, it is possible to open the other port and enter the next room.
If instead the atmospheric values coincide, the airlock procedure is not needed and the crossing is faster and more flexible. For this situation, a tunnel may be replaced by a simple door-type interface (such as the ones in ISS modules and its vehicles): the two bodies are at direct contact and no additional volume is added.
However, for completeness, this tunnel may also be *non-pressurized*, with just a simple passage connecting the rover to the habitat. However, it requires spacesuits, thus this trivial case is not to be considered. It indeed adds complexity: a direct link from the two modules is helpful only if allowing not to don the suits. Otherwise, a simple ground walk would be sufficient to go from the habitat's EVA airlock and the rover's one (or vice versa).

Many possibilities may be considered for the design, but two main elements are necessary, both on the rover and on the habitat: the *docking (or berthing) gear* and the *hatch*, which are exposed to the outside when not in use. The latter protects from depressurization when not docked.
It is also important to offer emergency back-ups in case of faults. One solution is having more than one docking mechanism, however causing a considerable weight increase. Another way is using the EVA suits to move from the rover to the habitat. This second possibility, as said, may also be the unique mode of access, in case design does not involve a direct docking passage (such as cars, not directly docked to the doors of our homes).
A last, and creative, solution is creating a big airlock-garage in the habitat. The rover thus enters in it, which then locks and pressurizes, allowing astronauts to walk and enter the habitat. As well as being not much practical, this solution is also critical for biological contamination.

In case of a more classical architecture, it is important to define its flexibility during the design process. It indeed may be compatible with other bodies, such another rover, a second habitat, or even an ascent module. This modular approach may lead to operational benefits.
Anyhow, the docking system is covered more in detail in this work.

EVA port

If the rover is designed to have an access to the outside, for scientific or maintenance Extra Vehicular Activities, or for the access of the rover itself, this passage counts as an interface with the outside environment. Many architectures may be examined[44]:

- *Single volume*: all the rover volume is de-pressurized for each EVA. This trivial architecture implicates all the astronauts in the rover mission must be simultaneously outside or inside of it.

- *Double volume*: a specific airlock room is dedicated to the EVA, being de-pressurized and re-pressurized at every cycle. For safety reasons, this environment must not be interposed between the main rover room and the possible docking unit (with the habitat), otherwise risking to block the astronauts in the rover from escaping in case of emergency.

- *Triple volume*: a third volume is added, for working in a de-pressurized environment. This unpressurised additional space may be used during the EVA for maintenance, recharging, item repository or similar functions. In case of contaminations and emergency, suits may be expelled.

- *Suitport architecture*: this solutions require the spacesuits being on the outside but directly accessible from the habitat, slipping inside. The movement is not simple in presence of gravity. Moreover, despite calling this case 'Airlockless', the interface between suits and rover requires a closure, not so different from an airlock: it is called *suitlock*. NASA Small Pressurized Rover demonstrator adopted this solution.

An important aspect is the protection from dust at the reentry. Dust is actually a problem for all the surfaces being both at contact with the outside and at the inside at different moments, thus all ports must be designed keeping this concern in mind.

Sample airlock

Since sample acquisition may be one of the main goals of the pressurized rover, a port for their entry is almost essential. This type of airlock may be small and require measures for contamination control. This interface may be seen as a simpler version of the one for EVAs.

Substances exchange with other modules

A pressurized rover may not be a self-sufficient entity. Thus, electricity, data, chemical components, fuel are some of the possible substances exchangeable with the habitat. This process may be done with specific components on the outside of the rover, or with specific interfaces in the docking unity (such as in the MGAAMA project[45]). Data and energy may also be exchanged wireless, generally with a lower rate.

3.3.2 Small Pressurized Rover

One of the most detailed demonstrative projects for a lunar pressurized rover was developed during the Constellation Program: the Small Pressurized Rover (SPR), which has also been tested in US desert.[45] However, it was not in its advanced design phases and was never pressurized.

Figure 3.11: Small Pressurized Rover (SPR) during desert tests.[45] Details of the docking flange are shown, with the male angular petals in black.

The mating system has been designed being passive on both extremes, in order to have the maximal usage flexibility: even two rovers could dock together, as well as with the habitat. This was possible thanks to the *Active-Active Mating Adapter (AAMA)*, an element with two active extremes, acting as an active pressurized connector between the two docking ports. It consists of a double Stewart platform, to give flexibility on 6 degrees of freedom, overtaking fine alignment problems between the two docked structures.
To allow the docking manoeuvre, the platform has to be already attached to one of the two bodies, or otherwise manipulated by a mechanical arm up to position. In the inside of AAMA, a simple plank represents the floor; it is manually positioned and retracted by the crew.
As perceivable from the CAD in figure 3.12, the soft docking system has four angular guides, compatible with the ones on the male docking unit of the rover (or habitat) seen in figure 3.11.

The advantage of this solution is mainly operational and maintenance flexibility; indeed, complex components are on the AAMA and not on the docking interface of rover and habitat. However, many weaknesses may be addressed, it being just a preliminary phase concept. Some are: low vertical space for walking, dust exposure during non-usage and docking practicality.

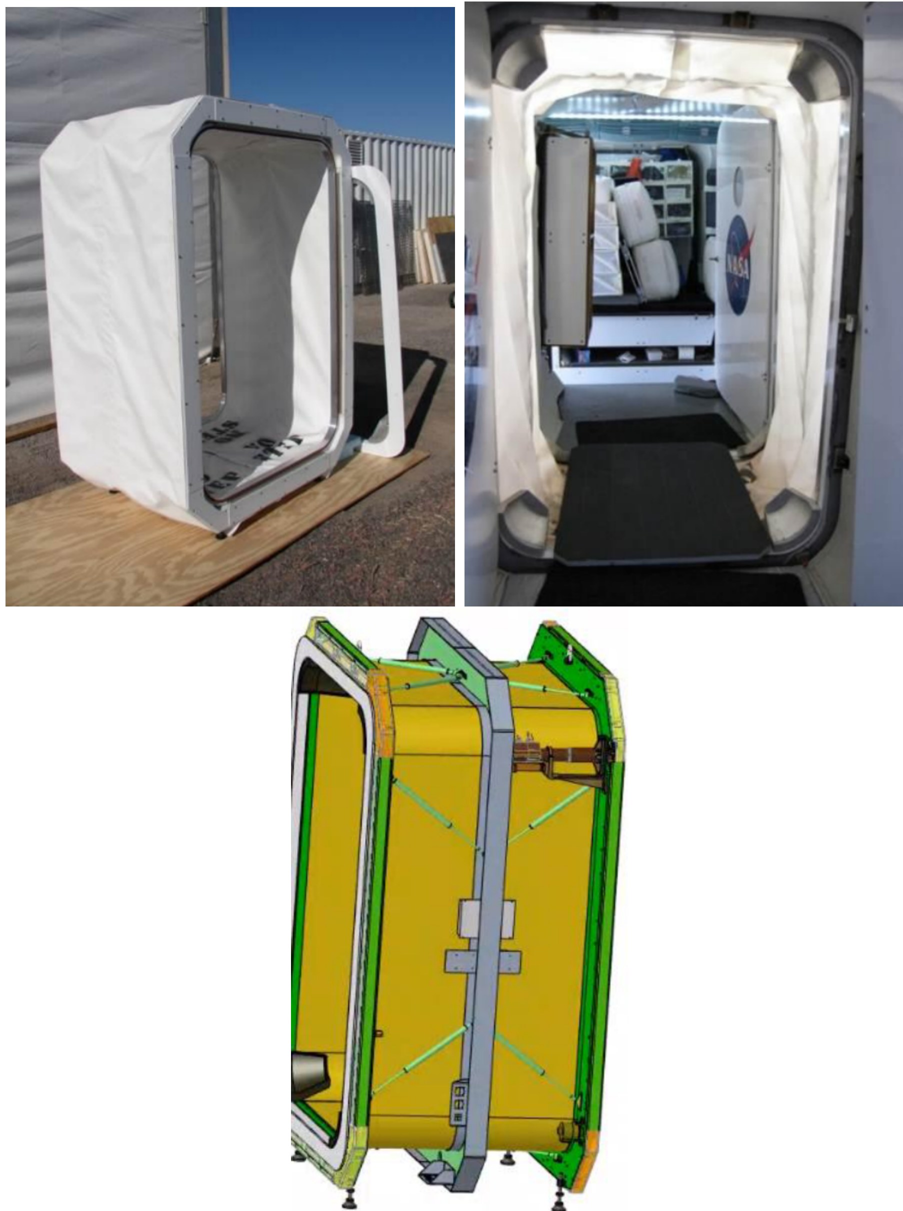

Figure 3.12: Pictures of the Active-Active Mating Adapter (AAMA) prototype. The two pictures above (external and internal) are from [46], the CAD model is from [45]. Poor quality of images is from their original source and does not derive from the reproduction process.

3.3.3 MGAAMA concept

An evolution of the AAMA platform, the Multi-Gravity Active-Active Mating Adapter (MGAAMA), has been proposed by Howard Jr.[45] It is presented as highly modular and suitable both for docking and berthing. It is again a double Stewart platform, circular

in section, with electrical actuators, allowing up to 15° of misalignment. This overcomes some strict alignment requirements, which are puzzling when on rough terrain and slopes.

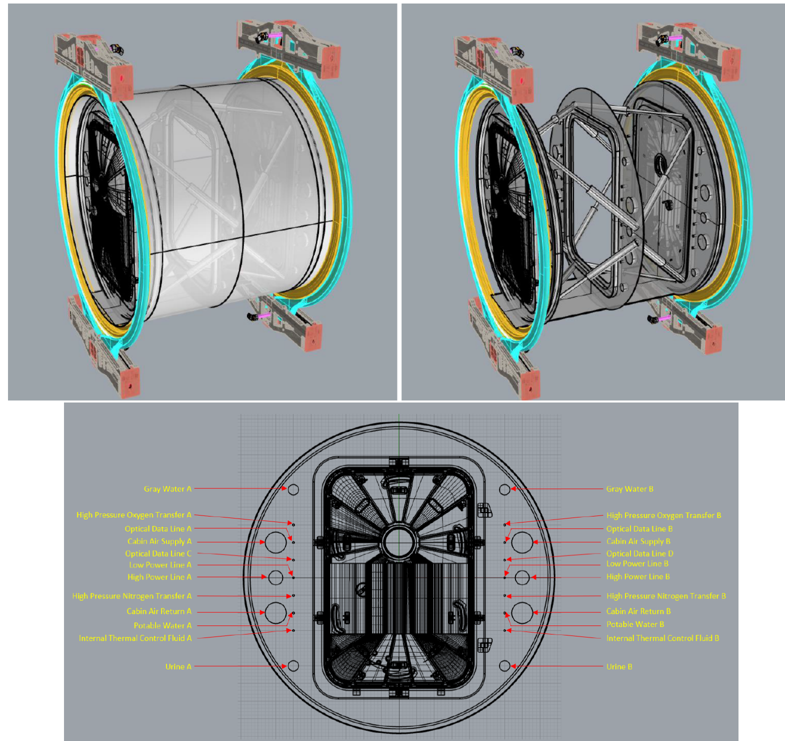

Figure 3.13: MGAAMA CAD model.[45]

Figure 3.13 shows the CAD model of the MGAAMA: one of the most interesting points of this design concept is the presence of multiple umbilical utilities transfers. In addition to crew transit, it may indeed accommodate the passage of gases, such as the ones for the ECLSS, water, electricity, data and so on. In different design choices, these umbilical connectors may be on the docking platform but radially outside the passage volume, or even allocated in different portions of the rover.
However, with the choice of allocating them inside the circular section, the hatch must then be smaller than the circular sectional area, as seen in the picture. It has so a rectangular shape, higher than wider to allow upright walk. The edges are blunted for better fatigue resistance, since lunar pressurized rovers are thought to serve for long stable periods with a reusable approach.
More ideas should be developed for protection against regolith dust and extreme temperature cycles, as well as for usability strategies.

Chapter 4

Requirements

The work starts defining the scope of the work, with a definition of the environment, of the overall project (namely the rover project), of the considered subsystem (the docking system) and the coordinates emerging from the previous work. Finally, requirements are listed, both for the system, both for the Digital Twin.

4.1 The lunar environment

One of the main challenges of the Lunar Pressurized Rover is the adaption to the Lunar environment, which strongly constraints the design. This brief review aims to introduces the aspects in which Moon harshly differs from Earth.[47]

4.1.1 Gravity

The Moon has a mass of $7.353 \cdot 10^2 2$ kilograms (roughly two orders of magnitude less than Earth) with a mean radius of 1738 km, compared to the 6371 km of the terrestrial one. It is, indeed, a mean value due to the flattening, which gives Moon an ellipsoid shape, caused by the attraction forces of Earth.
Noted these values, since the reciprocal gravitational force is given by

$$\mathbf{F} = -G\frac{m_1 m_2}{r^2}\hat{r}$$

with G gravitational constant, m_i mass of the considered body and r their distance, it is possible to compute the mean gravitational acceleration on lunar surface at Equator:

$$g_M = G\frac{m_M}{r^2_{Equator}} = 1.62\,\frac{m}{s^2}$$

This representative value is approximately a sixth of the terrestrial value of $9.81\,m/s^2$.

This point must be considered during the numerical sizing of lunar infrastructures, but should also be accomplished with in the ergonomical design. Indeed, Apollo astronauts showed a 'loping' gait more naturally than the classic walking. Moreover, steps are typically longer and hops higher then in higher gravity, resulting in non-mandatory constraints in dimensioning habitats and human-centred environments.

4.1.2 Atmosphere

If on Earth the standard atmosphere is of $2.5 \cdot 10^{19}$ molecules/cm^3 at sea level, on the Moon it is around 10^4 molecules/cm^3 by day and $2 \cdot 10^5$ molecules/cm^3 (newest works have suggested slightly higher values). These values, with a pressure of nano-Pascal scale, collocate lunar atmosphere as almost-vacuum.[48] Indeed, it is populated only by soil outgassing, particles coming space and their interaction with the ground.

4.1.3 Ionizing radiation

The thin atmosphere does not filter the various incoming radiation.[47] The main contributions are:

- Highly energetic *galactic cosmic rays*;
- *Solar wind*, with its large fluxes of particle possessing low energy;
- Strong sporadic fluxes of particles due to *solar flares*.

These particles are mainly protons and electrons. Solar components are highly variable, both on a small time scale in case of solar events, both on a large scale: the eleven-years solar cycle induces bigger fluxes during the maxima. The main footprint of these particles is on the health of organic bodies, ergo the astronauts, as well as potential plantations. Therefore, the Life Support System has to account for this and provide protection to organisms.
Some other types of radiation, such as gamma rays and electrons, are also present but less impacting and thus not better described here.

4.1.4 Lunar day cycle

The Moon spends around 29.5 days for a full rotation with respect to the Sun, as a composition of its revolution around Earth and the terrestrial revolution around Sun. Therefore, both day and night last about 14 terrestrial days. The day-night cycle must be obviously taken in consideration during the design phase: for thermic sizing, for mission planning and for solar energy production. Up to now, no astronaut has been on the surface during the night.
An exception to the 14 days changes is at the Poles: due to the small inclination of the lunar axis, many areas at the Poles are illuminated for long timespans, with the Sun appearing slightly over the horizon during many nights. On the contrary, also during the day the Sun does not rise over certain elevations. Therefore, deep craters never see the Sun on their bottom; such points, known as *Permanently shadowed regions*, are promising for the possible find of iced water deposits. This reason, and the long sunshine periods, are interesting features for future manned campaigns.

4.1.5 Temperatures

The just mentioned atmospheric thinness and day-cycle duration are responsible for wide temperature excursions: 280 K of interval between lunar noon and the coldest moment,

just before down. In details, it is mainly the absence of a robust atmosphere that does not damper the radiative heat fluxes between the Moon and the Sun (and, in a small percentage during the lunar night, of the lightened Earth). Approximate values for temperature are 100 K, as low boundary, and 390 K as upper one (indicatively -170^oC to 120^oC).
However, moving on Moon surface, the values may change. On the Equator, the average value is 255 K, with ±140 K of thermal excursion. However, at medium latitude, both the mean temperature both the interval value decrease, due to lower elevation of the Sun, which gives lower incidence and lower radiative energy. The extreme values are clearly met at the Poles, where the average temperature is only 220 K, with a low interval of ±10 K. The reason is, again, the low incidence of Sun rays. Finally, permanently shadowed regions show extremely low temperatures, probably around 40 K.
Another last relevant factor is the distance from the Sun during Earth's revolution, which gives about 6 K of difference, at noon, between perihelium and aphelium.

4.1.6 Soil and dust

Lunar soil, namely *regolith*, has subtle grains, comparable to silty sand. Their size is between 45 and 100 µm, with sharp and glassy grains analogues to many volcanic soils on Earth. Regolith density is around 1600 kg/m^3; other properties, helpful for terramechanical computations, are listed in [49]. More about trafficability is depicted in the next subsection.
Its grains' tiny dimension, aided by the low gravitational acceleration, cause regolith to cause serious concern in lunar missions. Also due to a facilitated electrostatic accumulation, it aggregates in clouds, which contaminate the interior of manned habitats. Moreover, the glassy shape adheres and attaches to surfaces, soiling tools, furniture, spacesuits and scratches biological tissues. Effects on astronauts include difficulty to breathe in contaminated indoor environments, obfuscated vision (due to suspended clouds) and lungs irritation.
One of the biggest challenges is to protect equipment from dust adhesion. This problem clearly concerns every tool which alternately stays both outside and inside, such as spacesuits, airlock hold doors and, thus, also portions of the docking gear: special attention should be posed on EVAs reentry procedures. Moreover, solar panels and fine mating parts, such as soft and hard docking components, must be safeguarded from dust accumulation and sandblasting effects. Thus, both protective and removing systems must be employed.

4.1.7 Trafficability

Trafficability is the ability of the soil to support a moving vehicle and to guarantee a satisfactory traction.[1] Apollo programme experience and unmanned rover missions revealed that negative estimates on trafficability were pessimistic. Indeed, wheeled vehicles well performed, with contact pressure not exceeding $7 \div 10\,kPa$. With various parameters known, or computed for the specific rover, energy consumption for locomotion may be estimated.

4.1.8 Meteoroid bombardment

Meteoroids and micrometeoroids (diameters smaller than 1 mm), represent a tangible risk on lunar surface.[47] Indeed, the almost null atmosphere is not sufficient to shield from these impacting bodies.
Impact probabilities generally decrease with the raise of the mass of the meteoroid; on the contrary, the harm is higher for bigger ones, which may perforate materials and sometimes be catastrophic. The risk is typically computed as a product of the two features, but is not easy and univocal to be assess. Estimates suggest micrometeoroids of milligrams of mass would strike a lunar habitat yearly, with higher frequency for smaller ones. The frequency increase for angles facing the Sun (diameters $< 1\,\mu m$) and following the direction of motion of Earth ($> 1\,\mu m$).
Velocities have been evaluated as $13 \div 18\,km/s$: microfractures have to be contemplated from micrometeoroids with masses of milligrams, especially in brittle materials as metal. A sufficient shielding may be offered by an energy-absorbing layer of $2 \div 3\,mm$ of a composite material.
Bigger meteoroids, such as 1 g of mass, may cause structural damages of centimetres. Chance esteems are of $\frac{1}{10^6 \div 10^8}$ yearly. More hazardous fractures may be assumed as bearable, with a real, but low, possibility of losing equipment. Such considerations should be reduced as much as possible for critical infrastructures, above all manned ones, which shall be strongly protected. Safety strategies are specific buffer protective layers, as well as placing such components sheltered between others, not to been struck.

4.1.9 Launch expense

The Moon is not just far from Earth. To reach it, it is mostly necessary to exit the potential well of terrestrial gravitational field. If for LEO missions many private enterprise offer solutions which may lead to a cost of $3000\,\$/kg$, for the Moon the expense is definitely higher.[38] Specific launchers need to be designed and configured, with a limited range of choice. Specific cost may rise up to more than a million of U.S. dollars for kg of payload.[50] Therefore, design should take into account this constraint, as well as the possible cost of launches back to Earth.

4.2 The Lunar Pressurized Rover

4.2.1 Space It Up programme

Italian Space Agency (ASI) and Italian Ministry for University and Research (MUR, Ministero dell'Università e della Ricerca) established *Space It Up* in 2024.[51] It is a Consortium which includes many stakeholders, thirty-three at the day of its constitution, such as Universities, public institutions and private companies Space it Up aims to develop new relevant technologies for the future Space industry and for its sustainability. In particular, it is centred on low TRL technologies, with a 2030-2040 horizon.
The research fields are organised in nine subcategories. Among these, *Spoke 1* concerns new missions for Earth's safeguard and sustainable development and planetary exploration;

Spoke 2 is for *Digital twins* and their applications; *Spoke 8* elaborates strategies for sustaining future manned missions; *Spoke 9* includes researches for travel toward and permanence on extraterrestrial bodies, with a natural focus on Moon and Mars.
Politecnico di Torino is one of the many universities and has also a central role in the coordination of the project. As a consequence, professors, young researchers, PhDs and students are involved in scientific activities. This thesis is thus integrated in the work of the project, in particular in the Lunar Pressurized Rover (LPR) working group.

Figure 4.1: Space it Up programme logo.[52]

4.2.2 The Lunar Pressurized Rover

As mentioned, one of the projects of Space It Up is the *Lunar Pressurized Rover (LPR)*, the preliminary design of a manned rover for future lunar exploration. This project is currently at a very preliminary phase, but it is not intended to go deeper in the very next years. The design target is to define possible brave architecture and technologies for more pragmatic projects in the following years. With this approach, the work is decomposed in sub-research groups for subsystems of the Rover. This work indeed collocates in the *Docking* and the *Digital Twin* topics.

From a conceptual view, the LPR is a manned rover aimed at supporting research activities and middle-long range trips on lunar surface, in the context of permanent presence of astronauts. It is intended to be a logistic facility as well as a travelling laboratory. Thus, it must be pressurized for safety and comfort. Consequently, this is a perspective coming not before a 5-10 years, even in the recent ambitious lunar exploration programs of the many Space actors, with the Artemis programme acting with a leading role.

Therefore, for the LPR it is acceptable to adopt immature promising technologies, trusting in their development in the next few years. However, there is a concrete target: the development of a scaled mock-up (1:3) to be tested on a terrestrial test-area. A local hypothesis is Mount Etna, in the South of Italy. To reach its , the work will gradually become more concrete and cohesive, while currently it sees many individual parametric and explorative works, such as Master thesis or small-groups researches, to be integrated

soon.

4.3 Why the docking system

This work has been centred on Digital Twin technology from the beginning. Similarly, it has been inserted in the Lunar Pressurized Rover context. The question was on which system focus the interest, since the main purpose is to draft a roadmap for creating a Digital Twin.
The first pick was the Environmental Control and Life Support System (ECLSS), being perfect for its continuous operations, allowing for a non-discrete data flow, elaboration and storage (perfect for creating an historical dataset). This type of system, composite, continuous and to be protected from anomalies, would be a well-fitting study-case to create a Digital Twin. However, for project necessities and mostly for the availability of previous concrete work, the focus has soon been moved onto the docking gear prototype.

But, why the docking system? A Lunar Pressurized Rover has never trodden lunar surface at present and the reasons are many. However, as a result, some necessary technologies have yet not been tested in space exploration history. Examples are a locomotion system capable of sustaining heavy weights, as well as the energy and power management. One of these novel technologies is the *docking* architecture between the rover and the lunar base (or possibly other bodies), allowing the transit of astronauts.
Never docking has been realized between a wheeled vehicle and a fixed one, with such levels of precision as required for space exploration: train gangways and trucks' loading bays do not meet such requirements. Also, microgravity docking is deeply different. This does not mean it is a never explored field either, but simply that conceived solutions never exited study stages and have never been tested in Space.
Thus, the docking mechanism study is one of the most stimulating design fields. It has to guarantee low-impact mating with the lunar base, hypothetically functioning for long operative time, ensuring long life to the structure. Moreover, it shall safeguard the crew from possible de-pressurization hazards in the most critical dynamic scenarios of the operations.

Nevertheless, the docking subsystem does not immediately well-fit the Digital Twin application. It has indeed an almost-discrete functioning, with fast and non-frequent activations, and faces a limited number of working cycles. Thus, the Digital Twin would be inactive for much of the time and, most importantly, it would be slow to collect data for statistical analyses.
For these reasons, and for the simple architecture required for such an academical work, the project has mostly directed to the analysis of the motion system for the docking prototype developed by Binetti. As said, the focus of the Digital Twin study has been mostly on the method, and less on the output (its functionalities). However, an integral Digital Twin for a Lunar Pressurized Rover (ignoring weight and computational limitations which would shrink the benefits and lead to a partial Digital Twin, applied only to some subsystems) would indeed require also the docking gear modelling. Some possible assets would be:

- Assist the design phase, from the choice of the architecture, up to the verification

phase;

- Helping the GNC algorithms to approach the habitat in a suitable way to facilitate the docking system reducing the misalignments;
- Collecting data to enhance future approaching strategies and new docking technologies;
- Individuating damages before complete breakage, protecting from potentially catastrophic events;
- Helping in maintenance phases, preventing the replacement of perfectly performing components;
- Offering a tool for facing possible emergencies, for example helping in the identification of suitable solutions.

For the current work, if by one side the choice of working on the docking system has been led by necessity, on the other it may indeed be an helping tool for the docking mechanism design itself, not only for the development of a Digital Twin implementation methodology.

4.4 Previous work

This section has the purpose of introducing the work of Binetti, which represents the basis of the current thesis for all concerning the docking system. To deepen these topics, please refer to the original Master thesis [1].

4.4.1 Lunar Pressurized Rover docking analysis

Binetti's work fits in the LPR project as one of the first theses activated in the program. It starts (*section I*) indeed with an overview of the rover, with a simple mission and system design of it. This analysis helps finding a first rough esteem of the global parameters of the rover, needed for more specific analyses such as the docking ones.

Section II starts with a review of docking from an historical and architectural review. From it, and the common source [38], section 3.2 was set up. Binetti then develops and explains mathematical models for the docking and for the contact mechanics, which is needed both for the terramechanics of rover's wheels, both for the alignment contacts of the docking surfaces.

From these considerations, he designs a first architecture of the rover's docking system, naming it CLASP (Circumferential Lunar Annular Mooring for Surface Pressurized operations docking subsystem). Requirements are defined, a trade study realized, as the focus is progressively moved to the three sections of CLASP:

- *Alignment system (ALN)*: the subsystem responsible for reaching a correct pose between target and chaser, during the approach. It passively guides the target interface up to complete contact with the chaser's one,

- *Soft Capture System (SCS)*: this subsystem gives a preliminary and not fixed, structural connection between the two interfaces.
- *Hard Capture System (HCS)*: stabilized the contact, sealing the two bodies hermetically, to form a single volume and allowing pressurization.

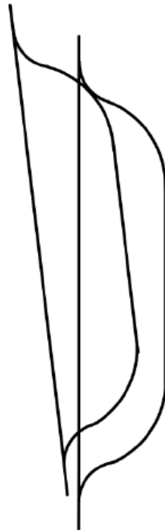

Figure 4.2: Sectional schematization of the double torus docking interface.[1]

It then follows a short description of the docking gear.
For the alignment, a *double torus* revolved structure, with a section similar to a bell's one, has been chosen as docking surface, both for the male (chaser) geometry, both for the female (target), as in figure 4.2. The surface is identical, to perfectly adhere. It thus is a peripheral, while not androgynous, passive system, with reduced complexity.
Since the rover approaches the habitat with limited precision, causing angular and translational misalignments, the target structure has to accommodate those. A simple mechanism has been chosen, with a set of springs and dampers behind the target interface, passively allowing for short movements to compensate misalignments with the chaser.
Compared to micro-gravity cases, the Soft Capture is simplified by the presence of gravity and terrain, thus slowing until contact may be achieved by wheel braking, which can also keep the position firm during operations. Such operations are easier and less resource-expensive then the micro-gravitational use of propellers.
The final step is Hard Docking, designed to be performed by twelve pairs of passive and active hooks, as in figure 4.3. Binetti explores a second strategy, less favourable, consisting in a set of magnets; one side has passive magnets, the other has active ones, electrically activated.

Binetti then explores each of these points in depth, illustrating the docking sequence and the design process of each subsystem, with proper rationales.
In *Chapter 10* he implements a digital model for the 1:1 scale rover, starting building the Solidworks© CAD model seen in figure 4.4. Both the rover's main body and the lunar

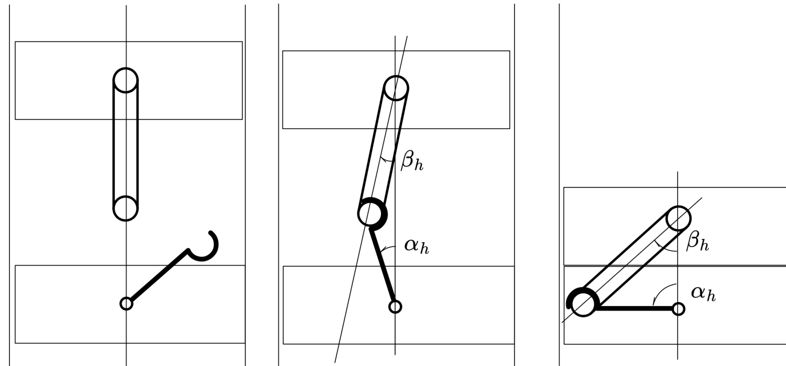

Figure 4.3: Pair of active and passive hooks adopted as Hard Docking mechanism. [1]

habitat are simplified as cylinders, since the aim of the work is not on their appearance or their functionalities; however, the model also aspires to preliminarily size the suspensions of the rover. With a proper dynamic motion definition, the model has been uploaded on MSC Adams ©, with a procedure similar to the one followed in this work.
Finally, a MATLAB © association with MSC Adams © was used to perform a numerical optimization, bringing to the final preliminary design.

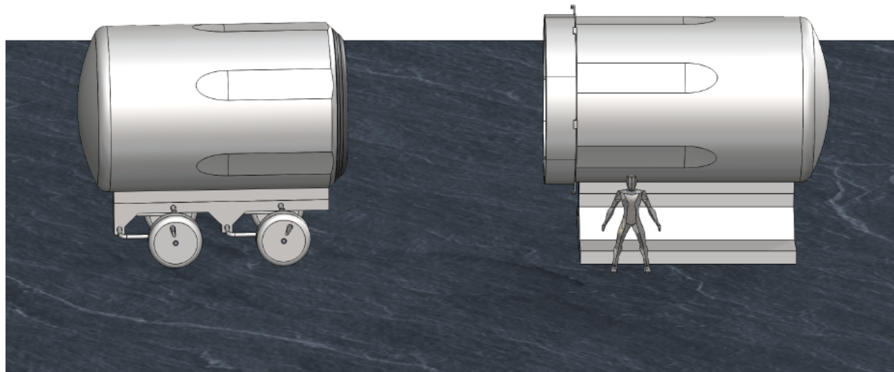

Figure 4.4: Visual CAD representation of the LPR and the lunar base, as in the CAD imagined by Binetti. [1]

4.4.2 The docking prototype

Part III deals with the prototype, developed as additional contribution in the Thesis work.
It is a 1:1 scale prototype of the docking interface with magnets, neglecting the structures around it (the habitat and the rover) and the Hard docking system, which aims to test the docking procedure. This section tries to resume the main elements in Binetti's dissertation. [1]

The docking procedure

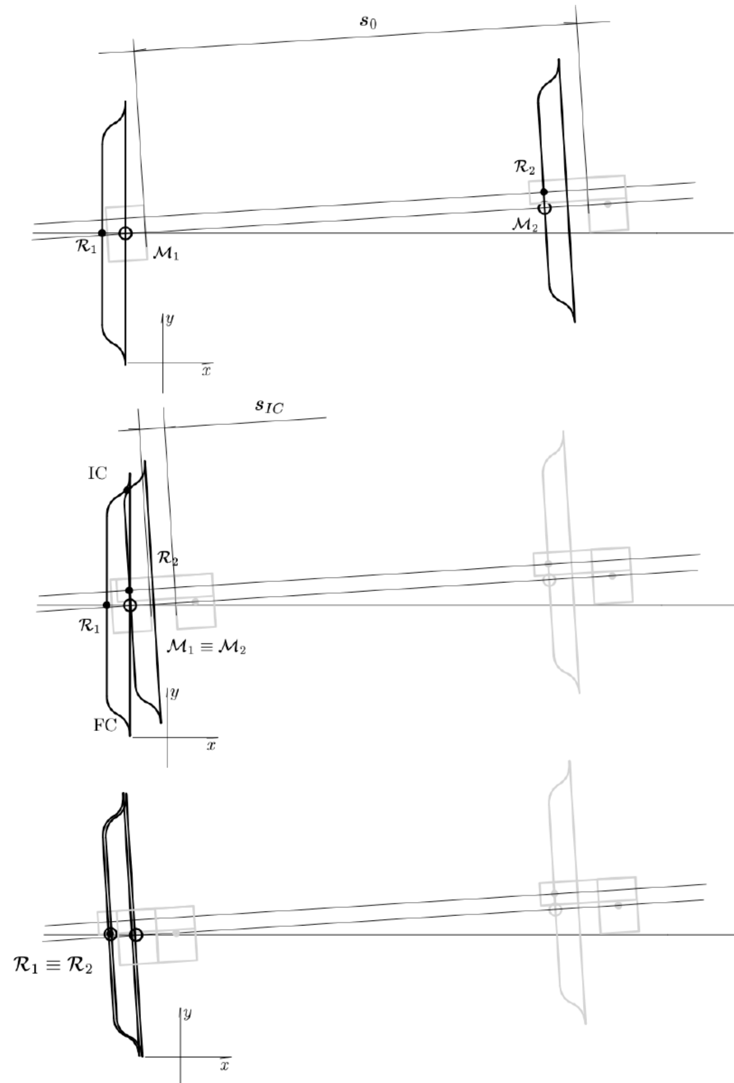

Figure 4.5: Formal docking procedure milestones, as defined by [1]. From top to bottom: initial state, initial contact, final contact.

The docking procedure contains three symbolical moments, here sketched and shown in fig .4.5.

- *Initial state*, at t_0: the initial condition considers null chaser velocity; it then accelerates towards the target, with possible misalignments.
- *Initial contact*, at t_1: the two alignment surfaces have the first contact, in any pair of points.

- *Final contact*, at t_f: the two alignment surfaces completely touch, adhering. From this moment on, the docking centres (see [1] for a formal definition) are aligned and coincident. The following and last phase would be Hard docking.

Test requirements

Along the line of those defined for the 1:1 docking design, a set of requirements has been defined for the prototype. Here, the main purpose is to help in the execution and evaluation of the experimental tests.
These requirements are listed in table 4.1, as from [1].

1:1 model

ID	Name	Symbol	Value	Unity	Description
P-R-PERF-B-05	Translational misalignment y	$y_d(t_1)$	± 0.1	m	At first contact, the Docking Subsystem shall be sized to sustain a relative translational misalignment along y of y_d
P-R-PERF-B-07	Relative translational velocity x	$\dot{x}_d(t_1)$	0.1	m/s	At first contact, the Docking Subsystem shall be sized to sustain a relative translational velocity along x of $\dot{x}_d$
R-PERF-B-10	Angular misalignment z (yaw)	$\phi_d(t_1)$	5	^{o}	At first contact, the Docking Subsystem shall be sized to sustain a relative angular misalignment along z of ϕ_d

1:10 prototype

ID	Name	Symbol	Value	Unity	Description
P-R-TEST-A-01	Translational misalignment y	$y_d(t_1)$	± 10	mm	At first contact, the Docking Subsystem shall be sized to sustain a relative translational misalignment along y of y_d
P-R-TEST-A-02	Relative translational velocity x	$\dot{x}_d(t_1)$	10	mm/s	At first contact, the Docking Subsystem shall be sized to sustain a relative translational velocity along x of $\dot{x}_d$
P-R-TEST-A-03	Angular misalignment z (yaw)	$\phi_d(t_1)$	5	^{o}	At first contact, the Docking Subsystem shall be sized to sustain a relative angular misalignment along z of ϕ_d

Table 4.1: Requirements from [1]: on top are shown three of the requirements used for the design of the 1:1 scale docking mechanism. On bottom, are listed the requirements of the 1:10 docking prototype, used to define the experimental tests.

Note that, as the horizontal plane confines the problem with respect to a free-space configuration (micro-gravity), the main degrees of freedom are three instead of six: two in-plane translations and one in-plane rotation. From those, the translational degree of freedom x is considered as velocity and not as displacement. The initial distance is indeed

not a problem, while at first contact it is null; what is of interest is the contact velocity, not to have a traumatic impact.

Test definition

The experimental campaign involves an approach and docking tests, combining different initial conditions, based on the requirements. Each of the three misalignments requirements (R-TEST-A-1 and R-TEST-A-3) in table 4.1 is taken null (x0, nominal values), halved (x$\frac{1}{2}$), as it is (x1) and doubled (x2). Such misalignments are inserted in the prototype with stepper motors and kept up to the final contact, as a GNC system was out-of-service.
Combination of misalignments values leads to nine test configurations. For each configuration, five tests were performed.

Definition of the prototype

Before designing the 1:10 scale prototype, Binetti had two options undergone a trade-off:

- *Vehicle model*: wheeled, it gives a better rendition of the behaviour of the real rover;
- *Railed model*: with more constraints, it has less resemblance but is easier both to build, both to control in misalignment tests.

Eventually, the *railed model* was selected for the prototype.

Description of the prototype

The docking interface includes:

- *Male and female interfaces*: double-torus surfaces in TPU (Thermoplastic polyurethane), with cavities for magnets;
- *Neodymium magnets* (x6) for docking force;
- Horizontal *longitudinal* (6x) and *and transversal springs* for female (target) interface motion and support.

Motion and misalignments control is possible thanks to:

- *Linear guide 1*: a linear actuator controlling the approaching velocity $\dot{x}$. It is kept approximately constant at 10 mm/s, with initial and final acceleration and deceleration ramps of 2 seconds, resulting in a trapezoidal profile. The linear guide has a length of $S_1 = 100\,mm$.
- *Linear guide 2*: a linear actuator for inserting the lateral translational misalignment Δ, which is conserved for all the trajectory. It indeed misaligns the whole *Linear guide 1* track, on which the chaser travels.
- A *stepper motor* inserting the angular misalignment γ.

The misalignments are shown in figure 4.6. Since y_d and ϕ_d, cited in the requirements (table 4.1) are evolving during the contact phase, from the contact to the final position, Δ and γ should be distinguished. They indeed are preset and conserved for all the approaching phase; to be accurate, Δ and γ are the misalignments between the chaser and the target support. During the contact, the target interface is slightly free to move around the springs, reducing y_d and ϕ_d to zero.
It should be noted that the *undocking* phase must be performed manually, since the magnets are all passive.

Figure 4.6: Misalignments introduced by the actuators. At top there is Δ, introduced by *Linear guide 1*. At bottom, it is the γ inserted by the *stepper motor*. [1]

The prototype works by controlling the actuators with an Arduino module and a set of batteries with a voltage divider. The two *linear guides* are actuators moved by stepper

motors, through a rotational-translational converting thread mechanisms. Binetti also reports the control algorithm used for Arduino. Details are in [1] in section 11.2.4.

CAD model

The thesis continues by reporting the CAD Binetti developed. In figure 4.7 it is shown an isometric view of the assembly. The main single parts drawings are also displayed, with some relevant dimensions.
The Bill of Materials is then reported.

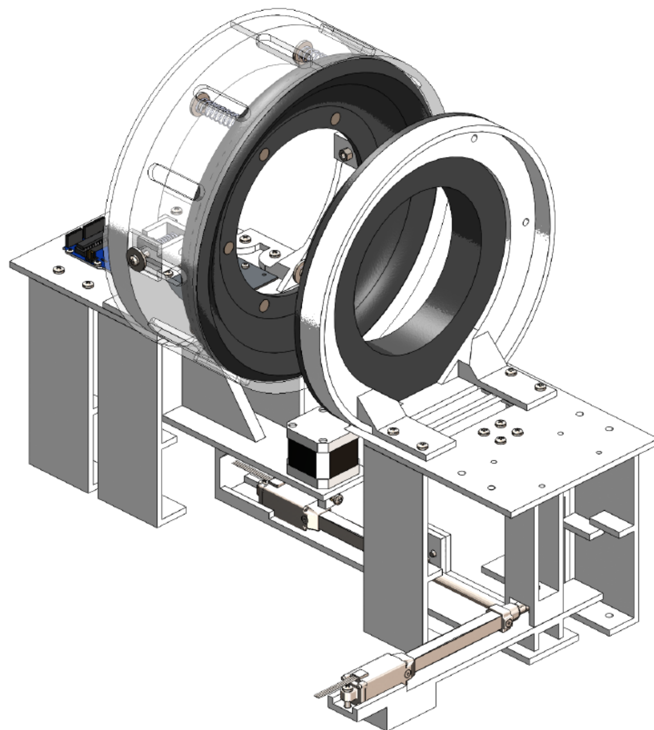

Figure 4.7: Isometric view of the prototype assembly, from [1].

Test results

Before the conclusions, Binetti reports the results of the tests, with a short comment. All the tests with null or, at most, nominal misalignments (as in section 4.4.2) were successful, for any combination of them. In particular, for each combination, five equal tests has been performed; the outcome has been considered as completely positive if at least four out of five were successful.
For higher, more stringent values of misalignment, tests were less favourable, with even two successes out of five trials for some combination. The prototype showed globally a good behaviour, not satisfying as it is for a real application, but fulfilling all the expectations for the work and representing a base for further design.

4.4.3 Minor hardware changes

During the recommissioning of the prototype, since it was disassembled for some months after Binetti's work, some issue emerged. Therefore, during the current work, some hardware adjustment was performed; these are here listed for further information:

- The lateral connection between the the halves of Vertical support and the target's main structure was blocked, not allowing for the expected small moves. The single screw linking the vertical support, target's structure (and an intermediate small part) has been replaced by two different screws with nuts and washers. The actual linkage is now performed only by a spring (*lateral spring*), which is simply wedged in by the washers at its extremities.
  This solution allow for small movements of the vertical supports, assisting the female interface in accommodating possible misalignments of the chaser.

- During he recommissioning of the electrical circuit, some of the short wires that connected the batteries to the voltage divider, and the voltage divider to the main board, broke. The broken cables had to be removed and new ones attached. Instead of a permanent linkage, longer wiring with intermediate connections was installed. This modular approach allows the batteries to be separated from the voltage divider, and the voltage divider to be separated from the circuit when turned off. Furthermore, the voltage divider is no longer suspended and held by electrical connectors, so the longer cables should experience less stress.

4.5 Digital Twin requirements

The current section starts to delineate the Digital Twin, by defining and indexing its requirements. Having indeed defined the scope of the work, the chapter ends by individuating the requirements of the Digital Twin, as section 3.1.4 suggests. This part of the work is one of the initial ones to be performed when starting to formalize the Digital Twin, aiming to delineate *what* it shall do. It indeed helps to formalize ideas, fixing targets for the design phase, before moving to the functional analysis.

About methodology, requirements emerged from ideas and notes taken during the previous phases. The brainstorming process was assisted by the guidelines of NASA about requirements writing. [53] Each requirement is justified by a rationale.
In particular, it should be noted that:

- *Shall* indicates a requirements;
- *Will* indicates a fact, or an anticipation of it;
- *Should* indicates a goal;

4.5.1 Layered model of Digital Twins functions

As depicted in section 3.1, a Digital Twin may show many uses and functions, showing various layers of complexity.

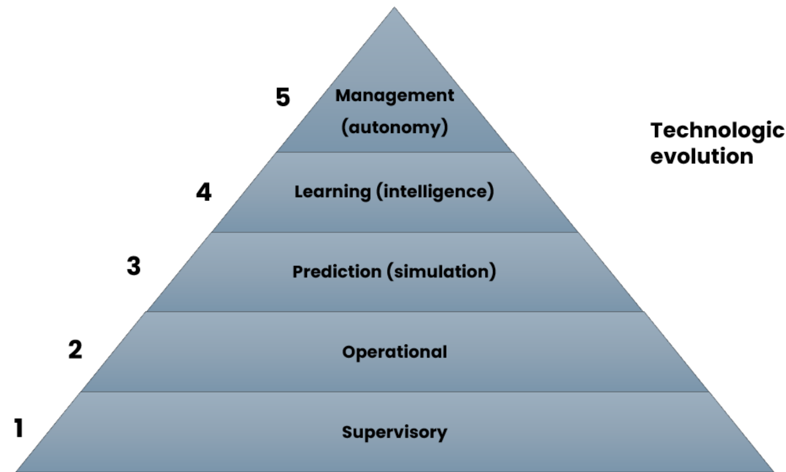

Figure 4.8: Levels of sophistication of a Digital Twin; based on [11].

Gardner et al. categorized the main different levels of sophistication and usage, as in figure 4.8. [11] These depends on technology maturity and request gradually increasing complexity, from bottom to top. Each level indeed also involves the implementation of the underlying ones.
The first two basic levels imply the possibility of assessing the state of the system and its operations, as well as sending commands in these scopes. Simulations predict the state of the system in different time spans, anticipating problems or contextualizing the span of choice to optimize decisions and operations. Finally, the two upper levels entail the use of Machine Learning and Artificial Intelligence to enhance the performance of the DT, up to independent decision-making ability to achieve full autonomy. Following the highest terrace of the pyramid, an autonomous Digital twin is therefore the most difficult to be designed, but also gives the biggest contribution in the management of the physical object, with the highest potentialities.

4.5.2 The requirements

Digital Twin requirements, listed here, must not be confused with the requirements of the prototype and those of the rover's docking system, both mentioned in subsection 4.1. [1] The analysis has been conducted by taking some sources as reference. [54] [53]
The *physical twin* is the physical system, in this work the docking prototype. Together, the physical twin and the digital twin compose a *Cyber-Physical System*.

Interface requirements

ID	Description	Rationale
R-INTF-1	At every time-step, the Digital Twin shall collect data from sensors.	The Digital Twin shall be able to obtain data from the physical system.
R-INTF-2	At every time-step, the Digital Twin shall collect data from the simulator(s).	The Digital Twin middleware shall be able to obtain results and data from the Digital system. It is the third layer of picture 4.8.
R-INTF-3	At every time-step, the Digital Twin shall store the status of the physical twin.	The Digital Twin shall create an accessible history of the system status.
R-INTF-4	At every time-step, the Digital Twin shall store the output of the simulation.	The Digital Twin shall create an accessible history of the simulation predictions.
R-INTF-5	The Digital Twin shall be able to recall the state of the physical twin of any time-step of the current simulation.	The Digital Twin shall be able to access the state of the physical twin and the simulation results at the previous steps, for comparisons, health monitoring or other purposes.
R-INTF-6	At each time-step, the Digital Twin shall send commands to the physical twin.	The Digital Twin shall be able to communicate with the physical body, establishing a bidirectional link, together with requirement R-INFC-1. It shall be able to send commands, to comply with the second layer of figure 4.8.
R-INTF-7	At each time-step, the middleware shall be able to send data to the simulator.	The middleware shall be able to communicate with the simulator to obtain status information and predictions.
R-INTF-8	An external user shall be able to run the physical twin by means of the Digital Twin.	There shall be an interface for users to operate the Digital Twin to start a run of the prototype.
R-INTF-9	An external user should be able to access the state of the physical twin at any moment.	There shall be an interface with which an external user be able to assess the state of the prototype, both current, both at past.

Table 4.2: Functional requirements of the Digital Twin, with rationales.

Performance requirements

ID	Description	Rationale
R-PERF-1	The Digital Twin shall guarantee a time-step smaller than 1 second.	The time-step shall be smaller than the time required for the full run of the prototype, to supervise a full run with more than one control point. The smaller the time-step is, the better it is for precision purpose.
R-PERF-2	At initial contact, the Digital Twin shall guarantee the system to comply the prototype thresholds requirements.	The Digital Twin shall operate as a motion controller, conforming to the first two levels of figure 4.8. The mentioned prototype requirements are those in table 4.1.
R-PERF-3	The Digital Twin shall allow to complete the docking procedure in standard operations.	The Digital Twin shall help the docking system to achieve the docking connection, despite faults of the prototype.

Table 4.3: Performance requirements of the Digital Twin, with rationales.

Functional requirements

ID	Description	Rationale
R-FUNC-1	The Digital Twin shall be modular to accept new sensors.	The Digital Twin shall be the most modular possible, allowing for possible updates; virtual sensors also may be considered. This feature is also called '*parametricity*'.
R-FUNC-2	At every time-step, the Digital Twin shall verify the presence or absence of faults in the physical twin.	One of the main functions of the Digital Twin is to assess the health status of the physical twin, with the goal of protecting from major and permanent faults. It recalls the pyramid basis in picture 4.8.
R-FUNC-3	The Digital Twin shall operate with at least two distinct physical twins by changing only the simulative model.	The Digital Twin shall be the most parametric possible, having an architecture adaptable to different physical bodies, at least allowing hardware updates.
R-FUNC-4	The Digital Twin shall take actions in case of faults to protect the physical twin.	To protect the physical twin, the Digital Twin shall stop the simulation in case of faults.

Table 4.4: Functional requirements of the Digital Twin, with rationales.

Chapter 5

Preliminary analysis

5.1 Model preliminary analysis

5.1.1 Functional analysis

Having the requirements cleared, the next step is to develop the functional analysis, to clarify which functions are required to the digital twin. In this work it is firstly debated about the *model* used to simulate the physical system, or *simulator*; the two terms are used equivalently. Then, section 5.2 faces the preliminary analysis of the actual*Digital Twin*. It has been chosen to differentiate the analysis in two sections in order to adequately select the model and the Digital Twin, separately, trying to better satisfy the requirements.

The *digital model* is the physical (here intended as Physics-based, not as concrete) or mathematical representation, virtually describing the physical body. It translates the relevant physical phenomena in the virtual environment, with a particular attention to evolution in time.

There exist no unique models. Since many may be adopted to represent the same phenomenon, this analysis aims to guide in the choice of the most appropriate.

The analysis has been performed by firstly decomposing the requirements, when they involve the digital model. As second step, a re-elaboration gave order to the considerations, in order to derive functions. The final step has been cataloguing and describing them, to have a formal and organized list of the functions. The result is the current section.

Simulatability

Simulatability is a neologism born as 'simulate' + 'ability', indicating the capability to perform a simulation or, conversely, to be represented by a simulation. The model shall be able to simulate the real system with adequate results.

- *To simulate the physical system*
    - *To represent the physical elements*
        - *To represent bodies*
            - *To represent volumes*

        · *To represent masses and inertiae*
        · *To represent contacts*
        · *To represent shapes*
        · *To represent positions*
        · *To represent poses*
      * *To represent the kinematics*
        · *To represent constraints and relations*
        · *To represent motions*
      * *To represent the dynamics*
        · *To represent dynamic actions (collisions also)*
  - *To predict future state of the system*
      * *To advance in time*
      * *To be a reliable model*
      * *To update the state of the system*

The decomposition considers both the capability to represent the physical twin, both to simulate its evolution time. At the level of representation, it shall define the features of the bodies, as well as kinematic and dynamic behaviour.
The second layer is the ability to advance in time to complete a simulation in time, propagating the state of the system.

Inputs

The second group of functions concerns the inputs arriving from the core of the Digital Twin, later called middleware. The model shall be able to manage them, automating the simulation and mirroring the real system at the beginning each time-step.

- *To manage inputs*
  - *To receive input data from external sources*
  - *To insert inputs in the simulation*
      * *To use velocity inputs as initial conditions*
      * *To use position inputs as initial conditions*
  - *To start the simulation by external commands*

Outputs

The third group analogously considers the outputs, with a focus on the ones required for prototype supervision. Moreover, it shall be able to send them to an external environment, that is the middleware of the Digital Twin.

- *To manage outputs*
  - *To produce outputs*
      * *To provide relative position data*

    * *To provide relative orientation data*
    * *To provide relative velocity data*
    * *To provide docking state data*
    * *To evaluate the distance from the theoretical docking*
  - *To send output data to external software*

Rapidity

To allow the Digital Twin to operate at least real-time (possibly hard real-time), the model shall simulate faster than the reality. As described in section 7.3, it is necessary to have

$$\alpha = \frac{t_{simulated}}{\tau_{clock-time}} \geq 1$$

In case this condition is not satisfied, the digital model strongly influences the Digital Twin, forcing the physical twin, if possible, to proceed at discrete interval (stopping each time-step to allow the Digital Twin to recover the lost time). It is intuitive that this functioning is not satisfactory and shall be avoided.
It should however be highlighted that the model by itself is not responsible of guaranteeing the aforementioned condition. Indeed, the hardware on which the simulator is run may be an obstacle. Increasing calculating capacities gives the model better performances.
In the scope of this work, the functions here listed will be tried to be satisfied; however, it is not mandatory, since the hardware is limited, being operated on an ordinary personal computer. Clearly, in case of real applications, this is not sufficient and also the hardware must be chosen to consent a full compliance (with margins) of such conditions.
Further considerations will be made later on, when the model is developed and its performances are known.

- *To be fast in simulating*
  - *To allow* $t_{simulated} > t_{real}$
  - *To be poor of superfluous elements*
  - *To allow for different simulation time steps*

Parametricity

A secondary-importance group of functions is *parametricity*. It is indeed not strictly compulsory for the operability of the Digital Twin, but gives it a flexible and strong framework. In detail, it means that the Digital Twin has modularity in itself, allowing for modifications; it is necessary to accomplish possible changes in the physical twin, as in case of updated design or of maintenance.
The highest level of parametricity would be a Digital Twin easily adaptable to very different physical twins. However, by definition, a model is suitable only for a restricted number of variants, since it has to describes it. Thus, in this section regarding the model, only minor changes and additions are considered.

- *To allow for modifications*
  - *To allow the addition of new elements*
    * *To allow new inputs*
    * *To allow new outputs*
    * *To allow new physical entities*
  - *To allow for modifications of existing elements*
    * *To allow to edit the requested inputs*
    * *To allow to edit the requested outputs*
    * *To allow to edit physical entities*

5.1.2 Model architecture

The *digital model* has to represent the physical model in the most accurate way possible, with the owed compromises to satisfy the functions just described.
The following step in the design procedure is thus to define its *architecture*. This step may be quite superfluous if the model is based on a commercial software, such in this work. Anyway, it helps clarifying the connections, helping in the construction of the model itself; so, it is not recommended to skip this passage.
Differently, the *architecture* is fundamental if the model is completely custom-made, especially when it is novel also the theory behind it.

The *architecture* considers the main elements of which the model is made of, their functioning and the connections. The level of detail is at discretion of the author, but at least one general architecture should be provided, with a limited detail level (to schematically depict the model in its completeness).

The diagram in figure 5.1 represents the architecture of the model. It is still at a quite abstract level, coming before the selection of the model. Here it is a brief description of it.
The *digital model* is only one, since only one physical simulation is needed: no multi-physical processes are involved. This simplifies the model.
The connections with the outside (the digital environment) are shown on the upper side, without specific details. It is not relevant where these connections lead, since it is considered in section 5.2.2 and is not a point relative to the model.

- *Inputs* enters the interest values to initialize the system and its simulations. As from section 5.1.1, some are position and velocity; other may follow. Also the command to initiate the simulation may be considered as an input.

- *Outputs* leave the model to reach other elements of the Digital Twin. They may be position, poses, velocity and various information about the state of the system and, possibly, of the simulation itself.

It is not fundamental to define precisely all inputs and outputs already, while it is to prearrange them.

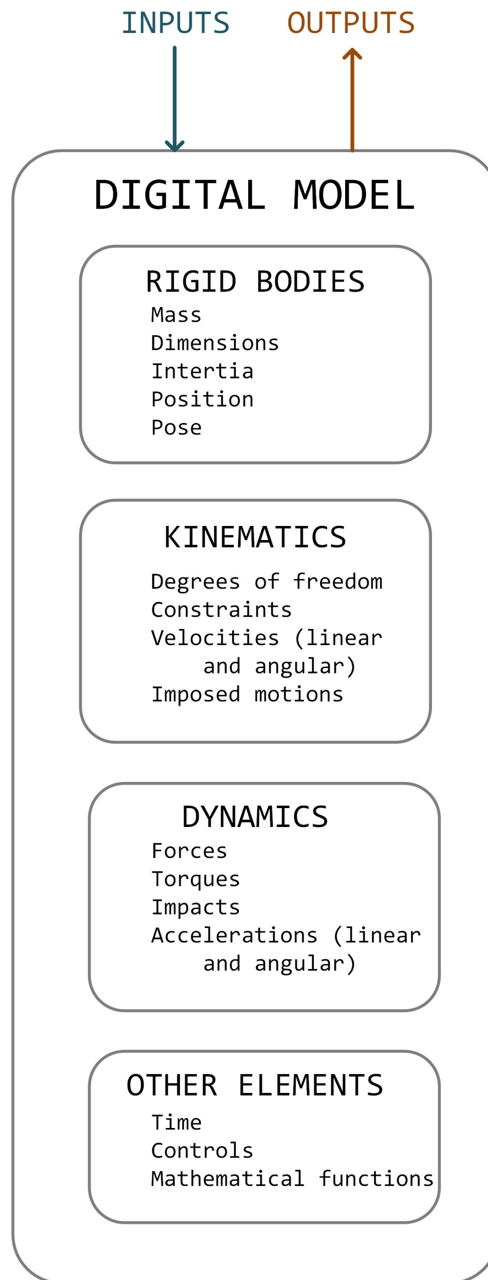

Figure 5.1: Architecture diagram of the model.

Inside the model, some blocks have been distinguished (from top to bottom):

- The model shall be able to represent the physical properties of the bodies. In particular, these have been chosen to be *rigid bodies*. In more advanced models, flexibility may be considered for critical elements: it would add information, but slowing down the simulation. Properties include: masses, inertiae, dimensions and

shapes, positions in space, poses (thus being able to represent systems of references), dimensions. Possible other elements may be surfaces, points, colours, and others.

- Another class of elements is needed to describe the *kinematics* of the model. Firstly, degrees of freedom and constraints shall be considered to represent the relations between bodies. Secondly, velocities and motions (imposed by actuators) are needed for representing the actions of non-blocked degrees of freedom.

- The most important elements for the analysis are the *dynamics* ones. Accelerations and inertial interactions, mostly forces and torques. Static actions are not sufficient, with the necessity to model impacts for the docking alignment.

- A final block contains mixed elements. One is *time*, which shall be representable to allow non-static simulations. To define complex models, the use of *mathematical functions* should be granted. Finally, the possibility to define *control* strategies is necessary to enhance the complexity of simulations.

5.1.3 Study cases

The next step is to define the study cases. This part of the preliminary analysis is useful to narrow the scope in which the model should operate, in order to select it.
Possible broadenings may be introduced later; however, adopting from the beginning a flexible model is valuable, to allow such scope variations with different study cases. A compromise between flexibility and complexity should be taken, depending on the purpose of the project.

For the prototype, and thus its digital model, the study cases initially considered are the original ones from Binetti's work.

- The first one is the *nominal case*, wit null misalignments. The Chaser moves from a distance of 100 mm (between the centres of Target and Chase adhering surfaces), starting with zero velocity. The speed profile is trapezoidal, with a constant velocity of 10 mm/s. The transition phases, both the initial acceleration, both the final deceleration, are 2 seconds long, as in figure 5.2.

- A second study case is the one with threshold requirements misalignments, as in table 4.1; it may be called the *requirement case*. The linear guide, with the same velocity profile as before, is misaligned of $\Delta = 10\,mm$ and $\gamma = 5^o$. With this conditions, all tests were successful in the work of Binetti.

- A third study case is the most ambitious one, with doubled values of misalignments: $\Delta = 20\,mm$ and $\gamma = 10^o$. This situation, the *double requirements case*, is the most critical among the tested ones: Binetti reported partial success with such requirements. The speed profile is again the one in figure 5.2.

All these cases are in turn included in *standard operations*, meaning that no fault shows. Malfunctionings may indeed cause failure in docking procedure. If, by one side, resilience to faults should be implemented in more advanced design, with reliable and

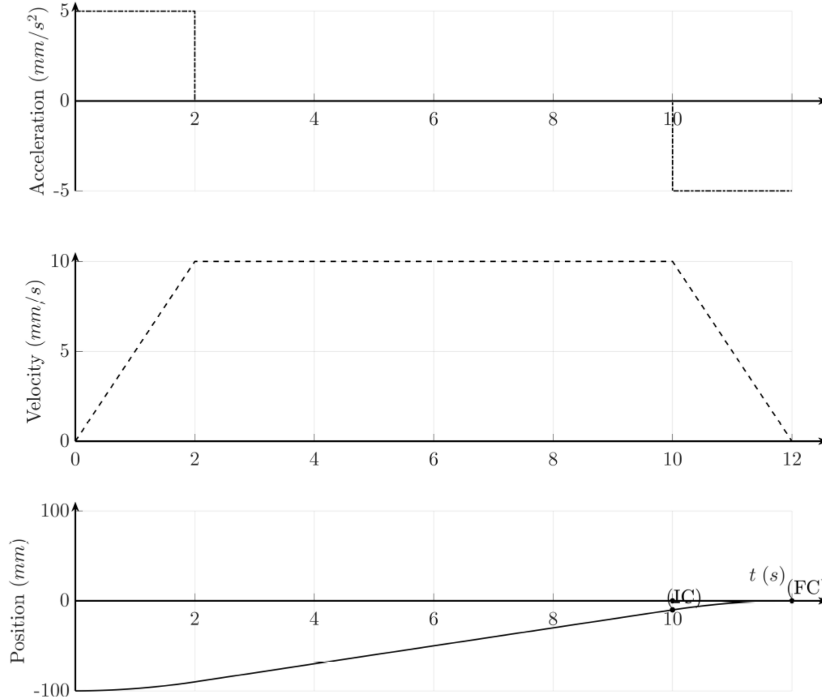

Figure 5.2: Nominal acceleration, speed and position profiles, imposed by Linear Guide 1, as from Binetti's thesis. [1]

redundant technologies, such a situation is tolerated in this phase of the design, not strictly limiting the purpose of this work.

Possible additional study cases, as mentioned, may be implemented at the achievement of the main purpose of the work. Examples may be the combination of positive Δ and negative γ (or vice versa), bigger misalignments, different velocity profiles. Drastically different cases may require alterations to the model.

Context may be added by discussing such study cases related to the hypothetical Lunar Pressurized Rover. They represent indeed strong limitations: the design of the docking interface is still preliminary and does not contain redundancies, as mentioned. Moreover, many technologies, such passive magnets for Hard Docking, are not adequate to real-scale applications. In addition, the railed motion system introduces strong simplifications. Some are by-passable, such as the velocity profile; others are not, such the absence of vertical misalignments, which may be present for terrain undulations or uneven weight distribution on the suspension system. These second order misalignments must be considered during the design phase, differently from this current work which individuates only the three main degrees of freedom (two translations and one rotation, all on the ground plane).
On the other hand, the real rover shall be equipped with powerful control (also human guidance) and GNC systems, with the Digital Twin possibly constituting one of such tools. These utilities would contribute to reduce misalignments to small value; however, their faults shall not exclude docking capability.
Balancing such considerations, the considered study cases are nonetheless of interest, since

the requirements were derived from literature, as justified by Binetti.[1] Moreover, the scenario with doubled misalignments inserts an high stress element on the hardware. However, the real tolerance to misalignments and the achievable precision of approaching manoeuvre shall be carefully studied all over the design process of the rover, with a special attention to physical tests.

5.1.4 Model selection

What models are

Before discussing the choice of the model in this work, it is useful to introduce the concept of model.
A *model* is a representation of a physical phenomenon, containing and elaborating information and data about it. Generally a model is derived from theory and described with mathematical means. Models may actually be physical (concrete objects), such as scaled mock-ups used for testing, or abstract, based on concepts and mathematical tools. In this work, the term *model* refers to the second case.

Models are not univocal, since multiple models may be developed to explain or simulate the same phenomenon, perhaps with different assumptions and precision. Moreover, since mathematics is just a language, there are different ways to represent it. Take, as trivial example, a body falling without friction; it may be treated both with a force and acceleration, both with a energetical approach.
Secondly, models are almost never exact, again being just representations of phenomenon. They seem exact when validated and respecting their foundational assumptions; but rarely a phenomenon is such to be perfectly described by a model without simplifications or uncertainties. Validation is done in fact net of an acceptable tolerance, since the real world is generally more complex than representable in a model.
It should not be forgotten either that science is always open to the possibility to confute what is validated until then.

Another limitation is that models may be indefinitely expanded, by adding different layers, interacting phenomena or even phenomena emerging at higher (or lower) scales of magnitude (such as a molecular/atomic level analysis of friction inside a macro-scale problem).
No model is thus 'complete', since it is the user to decide whether if it satisfactory. Indeed, after delineating purpose, scope and desired accuracy, it is possible to speak of completeness as the feature of satisfying all the requirements. In such a case, it is always valuable to adopt the simplest model among the possible ones, to avoid superfluous complexity.
However, it should be borne in mind that there are rules to follow for adequate validation.

This work considers models as *numerical* or *digital*, more than *mathematical* and *analytical*, even if sometimes the difference is minimal. Indeed, phenomena shall be described with a language readable and executed digitally by a computer.
The difference between digital and mathematical models is in the language. If simple mathematical operations such are univocal, it is not the case of non-linear or differential equations, or even the treatment of irrational numbers and digits. Then, just the

introduction of a method to solve these equation creates a discrepancy between the pure mathematical model (often already different from the pure physical phenomenon) and this numerical one, with unavoidable approximations. The digital need of discretizing continuous fields also introduces approximations and errors.
It is then clear how the number of different variants for a model is almost infinite: it is therefore crucial to choose the simplest ensuring the required precision.

Classification of digital models

Entering in the field of digital models, Segovia and Garcia-Alfaro individuated a distinction from the point of view of building a Digital Twin. [54] It is already a higher-level and engineering analysis, since the focus is not only on how to describe the system mathematically, but also on how to organize it.
Their classification starts separating between *Structural models*, which specify how the system is assembled and the relations between the components and its structure, and *behavioural models*, which describe how the physical entity works.
In details, the authors distinguish *structural models* into:

- *Physical models*, representing some of the phenomena occurring on the body (or the system) and some of its properties;
- *Geometrical models*, outlining geometrical features, kinematic behaviour and system's interfaces; a CAD model is an example.

The *Behavioural models* are discriminated as:

- *Control models*, based on control theory, physical laws and known information such as mathematical models. They are more than descriptive models, using comparisons between data and simulated results to elaborate commands for the system; Kalman and particle filters may be implemented for precise estimations;
- *Data-Dependent models*, a recent approach based on data structures; these models work as black boxes and are often powered by artificial intelligence or machine learning;
- *Hybrid Control-Data models*, which combines the features of the two aforementioned ones, such as physics-informed machine learning;
- *Other models*, exploiting relations between components, such as graph-depending models.

Surrogate models

A brief parenthesis should be dedicated to *surrogate models*, sometimes called *metamodels*.[55] Their relevance emerges when actual models are numerically costly. Indeed, a great number of variables, combined with high fidelity, require long time to simulate high-fidelity models.
While long computational time is acceptable for scientific simulations or preliminary design

optimization, it is an enemy when the model should be employed in automatic procedures, especially when real-time is the focus. new results, the situation may change for Design of Experiments (DOE) and optimizations, and gets unbearable for real-time applications.
Despite the increase of computational capacities and the development of improving methods such as parallel computing, often it cannot keep up with the enlargement of the models. Thus, a compromise emerges: *surrogate models* approximate the original models with simpler methods, with the goal of drastically reducing simulation time with the smallest possible loss of accuracy.

The process behind the development of a surrogate model starts from an analysis of the model variables. Highly-relevant variables are identified as inputs for the surrogate model, which aims at easily obtaining an estimation of the outputs from a set of inputs. From the identified variables it is thus sketched a Design Of Experiment (DOE), to have a set of simulations of the full model, with the most heterogeneity of the input variables, each with a reference limit range, that should be properly defined on the base of the phenomenon of interest. The high-fidelity simulations results are used as starting point to develop the surrogate model, from simple interpolation techniques to complex scientifically informed machine-learning models (provided that the resulting model is smoother than the replaced complete one.

Techniques are many and varied, from Response Surface Method (RSM) to adaptive learning methods, as shown in [55]. As already mentioned, the two indicators are: *computational time*, which is the time to obtain the result data of the simulation, and *accuracy*, the closeness of a result value to a standard one.

Just as example, something is said about *Response surfaces method (RSM)*: assuming an independent vector of variables (some may be chosen to be fixed, some may be varying inputs), the response vector is y: $y = f(x) + \epsilon$, with ϵ the random error, normally distributed with a mean of zero (and zero standard deviation). [55]
The RSM approximates the response, which is otherwise obtained with the complete model, with a surrogate function: $\hat{y} = g(x)$.
One simple function for $g(x)$ is a linear function, which gieves:

$$y = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_i x_i + e$$

Higher-order polynomial can be adopted for better results, especially for non-linear system. Therefore, RSM is a simple method, giving good performance for non big sets of variables and for linear functions. With other conditions, many other methods produce better results.

Tools for modelling

As the digital world evolved consistently in the last decades, many software tools emerged to assist engineers in their work. More recently, in the last decade Digital Twins established themselves, as Tao et al. pointed out in their review [5]. They analysed both literature and commercial ventures, since enterprises aligned on this trend by releasing their own suites for Digital twin development.[56]
Examples of software that help in developing models and, consequently, Digital Twins are: [5]

- CAD software: CATIA, SolidWorks;
- Structural engineering tools: Ansys (various physical fields), Abaqus (mostly for non-linear analysis);
- Various components systems (multi-body and multi-physical analyses): Simulink, OpenModelica, MapleSim, Adams.

The list is clearly not exhaustive, being just a glimpse of the possible commercial software tools, with a focus on the area of interest of this work.

Other recent technologies combining hardware and software are improving faithful modelling capabilities. An example is *point cloud technology*: with scanners, it is possible to create a loyal 3D virtual copy of a body. Such an accurate duplicate can be coupled with a Physics-defined model for extremely precise simulations. [5] Again, this is just one of the many possibilities offered by technologies arisen in the last years.

Nevertheless, it must not be forgotten that models can be *custom-made*. It may be built with an advanced high-level language, such as Matlab or Python, intermediate as Fortran, or even low-level language such as Assembly.

Model choice

Having performed the preliminary analysis, it is time to choose among all the possible model types, before building it complying with the functional analysis and the architecture defined.

The system to be simulated, the prototype, is an *electro-mechanical multi-body* system. The electric and electronic components are neglected , with only their effects considered. It means that they are not modelled, while the resultant motions are directly included in the simulations. Therefore, the prototype is just considered as a *mechanical multi-body* system.
Such system do have a wide range of commercial and open-source alternatives for modelling purposes. It is then worthless to build a *custom model*: the effort to define all the elements identified in section 5.1.2 would be remarkable, without robust advantages. The first choice is thus to use an already existing software tool.
Furthermore, the prototype has mostly been crafted by 3D printing. Consequently, there already existed a CAD version of it, a valuable framework to build over.

The choice therefore remains between*commercial* and *open-source* software. For the first category, Politecnico di Torino offers licence to Exagon suite, which contains MSC Adams, an analysis tool for multi-body problems.
The choice fell on *MSC Adams* for some reasons:

- Open-source applications are usually more complex to be used, while allow more flexibility and possibilities. Here, the system to be modelled is quite simple, without advanced concepts, and defined. Therefore, commercial software is the better option, requiring less effort for training and development.
- MSC Adams already has interfaces with other file formats and software. One is the possibility to import a CAD model to start the project; it also offers integrations

with MATLAB/Simulink, considered candidates for the Digital Twin core. Also, other programs of the same commercial suite open the possibility to perform further advanced analyses in a second moment, expanding the model on different areas.

- Another advantage of Adams is that a relevant layer of the model has validation in itself. The mathematical language behind representation of physical elements and the methods used for numerical solutions are intrinsically enclosed in the software and do not require any further validation.
  What, instead, still requires validation, is the representation of the real system inside the model, since all elements shall mimic the real system. A trivial example: if the mathematical model behind the software is correct but the rover is represented with squared wheels on the roof and horizontal gravity, the model is clearly not correct. Not using a custom model therefore allows to validate less layers, reducing the effort, but it must not taken as true without a comparison with the real system.

Considerations

Some considerations should be made regarding how MSC Adams aligns with the preliminary analysis performed; these observations replace the description of how the model represents the reality, since it is not custom-made. Instead, a description of how the model is built in MSC Adams is in section 6.3.
Recalling the functional analysis in section 5.1.1:

- From *Simulatabiliy*, the capability to represent the physical elements is completely satisfied by Adams, being a platform to represent multi-body mechanical systems, consolidated by years of operational use. Analogously, the first three blocks in figure 5.1, with bodies properties, kinematics and dynamics, are representable.

- Again from *Simulatabiliy*, the capabilities to advance in time and to update the state of the system are granted, since Adams is able to simulate in time.

- A last function from the same subsection, is to be a reliable model. Supposing it correctly represents the reality (with the model well built by the user), Adams is considerable a strong reliable model. Indeed, it has a long history of applications, both in academia and industry. It is therefore considered more than validated by use.

- The functions in *Inputs* section are all satisfied, with many possibilities to externally control the simulation. The received inputs may be inserted in the simulation as data.

- Similar considerations hold for the *Outputs* section; in particular, Adams is able to generate such types of outputs with its simulations.

- Skipping to *Parametricity* functions, all may be satisfied by simply modifying the model in the command file or with the graphical interface. If the model is exported in other formats, it is necessary to come back to the model and re-export it.

- Returning to *Rapidity*, the discussion is more critical. Indeed, model simulation speed depends: on the used solutors (the code is run in C++ or Fortran); on the built model (a more complex model would be more realistic, but would also slow down the simulation); on the hardware itself. Therefore, the question of whether the model is able *'To allow* $t_{simulated} > t_{real}$*'* is not univocal and depends on the model itself and not only on the type of model. Adams does not guarantee it by itself.
  If, however, this point is satisfied, the model can also variate its time step, real-time and non-real-time (faster, up to the performance limit, and slower than real-time).

5.2 Digital Twin preliminary analysis

Moving to the preliminary analysis of the Digital Twin, the steps are the same as those just been covered: functional analysis, architecture definition and software selection. Such process helps in the definition of the Digital Twin, allowing to methodically start its creation.

5.2.1 Functional analysis

The first step is to define the functions the Digital Twin has to satisfy. The used method was the same described in section 5.1.1 for the model.
The analysis started with an accurate examination of the requirements and the decomposition of the deduced functions. These were then arranged into categories with a logical hierarchy. This is followed by the list of functions with a short contextualization.

Decision making

One of the main groups of high-level functions is the capability to take decisions. For this work, this ability is strictly deterministic, but machine learning methods or any other complex algorithm may be implemented for better results; these would indeed insert a more flexible and powerful element in the Digital Twin.
The functions are mainly: to compare the state of the system with the desired one, as in a simple control algorithm; to use such decisions to implement the control strategy, by elaborating commands to send to the prototype, written in an executable language.

- *To ensure the prototype undergoes initial contact with* Δ *below a specified value (in standard operations)*
  - *To compare* Δ *value with the desired one, at each time-step*
  - *To compare* Δ *value with the one predicted at the beginning of the previous time-step*
  - *To decide to send commands to vary the* Δ *misalignment*
    - *To decide new values for* Δ *to be imposed*
    - *To convert such values into a command readable by the prototype*

- *To ensure the prototype undergoes initial contact with γ below a specified value (in standard operations)*
  - *To compare γ value with the desired one, at each time-step*
  - *To compare γ value with the one predicted at the beginning of the previous time-step*
  - *To decide to send commands to vary the γ misalignment*
    - *To decide new values for γ to be imposed*
    - *To convert such values into a command readable by the prototype*
- *To ensure that the chaser follows a predetermined speed profile (in standard operations)*
  - *To compare speed value with the desired one, at each time-step*
  - *To compare speed value with the one predicted at the previous time-step*
  - *To decide to send commands to vary the speed*

Health monitoring

The Digital Twin has to supervise the health status of the prototype, by checking if the values provided by sensors are compatible with standard operations and with the predicted values. In case of hypothesized faults, the Digital Twin shall interrupt the simulation by elaborating a command readable by the prototype.

- *To verify the presence of faults*
  - *To compare the measured values with the ones predicted by the last simulation (if no new commands were sent and executed in between)*
    - *To decide if a discrepancy in values is associable to a fault*
  - *To compare system's state with fault values*
    - *To compare speed value with fault values (too high, null)*
- *To interrupt the run in case of fault*
  - *To elaborate a command executable by the prototype*
  - *To send the command to the prototype*

Sensors integration

Since sensors are not already implemented in the prototype, they shall be separately integrated with the Digital Twin. At this stage, the sensors are indeed considered separately from the prototype.
The possibility to add new sensors or to combine existing ones into new virtual sensors shall be considered. Also filtering is considered, in order to clean data from the sensors. Such a method may deeply vary from case to case, and thus the function is kept general;

for instance, in refined versions Kalman and particle filters may substitute simpler filters. A more detailed analysis is in section 6.1.

- *To integrate sensors*
- *To allow the definition of new virtual sensors*
- *To integrate new sensors*
- *To integrate virtual sensors*
- *To filter sensors data to reduce noise impact*

Storability

Another important point is the capability to store and recall old data. The function 'to collocate stored data in time' means that data shall be associate to an instant, by using a specific system (absolute time, such as the time of the day, or relative time since the beginning of the simulation), with an approach similar to metadata.
For the reading and recalling, all the mentioned types of data shall be considered (system's state, sensors data, simulations, commands).
The function 'to read data of the old time-steps $(t - k\Delta t)$' may not be observed in the first version of the Digital Twin. It does indeed add a little complexity, while being useful mostly for better filtering (e.g. Kalman algorithms) or for advanced approaches such machine learning and AI algorithms.

- *To store data*
  - *To store data about the system's state*
  - *To store data from sensors*
  - *To store data of the simulations*
  - *To store data about commands*
  - *To collocate stored data in time*
- *To read data from the storage*
  - *To read data of the previous time-step* $(t - \Delta t)$ *from the storage*
  - *To read data of older time-steps* $(t - k\Delta t)$ *from the storage*

Operations management

This group includes general functions behind the co-simulative management of time and processes. The topic will be better covered in 7.3, so these functions are not completely developed in the preliminary phase.

- *To advance in time (based on the computer clock)*
  - *To control time-step being the desired one*
    - *To stop the run if the time-step is not achievable by the simulation*

        * *To report unfulfilled time-step to the user*

- *To coordinate activities and signals in a proper sequence*

Human interface

Human interface is necessary to impose, at the beginning of a run, the desired state of the system, i.e. the position and speed threshold values within which the prototype must be at first contact, such values replace the default ones, that are the ones in the requirements. It may also be considered to insert the initial state of the prototype, despite sensors should be enough to assess it; therefore, such function has not been included.
Moreover, the user shall give the command to start the prototype's run, being this project just intended for isolated tests. In case of a nominally operating real docking system, the Digital Twin may be always active, or at least a strategy for its states (such as active, off and stand-by) must be sketched.

- *To insert the desired values of the starting simulation*
    - *To insert the desired* Δ *value (at initial contact)*
    - *To insert the desired* γ *value (at initial contact)*
    - *To insert the desired speed profile*
        * *To insert the peak speed*
        * *To insert the duration of acceleration and deceleration phases*
    - *To insert the time-step*
- *To insert the command to start a new run*
- *To read the state of the simulation*
    - *To allow the user to request data*
    - *To print data (relative to the instant they were required)*

Inputs and outputs

Interfaces of the middleware include: the prototype, with Arduino-governed actuators; the sensors; the simulator (which is part of the Digital Twin); the user; the storage. Some of these points have been treated separately.
The analysis may be done more in details studying all the different interfaces. In particular, the middleware shall be able to dialogue with MSC Adams, selected to build the model, with Arduino Uno, which controls the prototype, and with the sensors.

- *To obtain data from the sensors*
    - *To ask data to the sensors*
- *To obtain data from the simulator(s)*
    - *To ask data to the simulator(s)*

- *To send data to the simulator(s)*
  - *To send initial conditions data*
  - *To send the command to start the simulation*
  - *To send the required simulated time-step*
- *To send commands to Linear Guide 1 (speed value)*
- *To send commands to Linear Guide 2 (Δ value)*
- *To send commands to the Stepper Motor (γ value)*
- *To send commands to start the run to the actuators*

Advanced functions

This short section covers additional functions that were not included in this prototype. These functionalities would introduce introduce complexity, while leveraging the quality of the Digital Twin. However, not being essential, they have been discarded from the first model, with the intent to include them in the first possible updates.

- *Time-step adaptability*: the capability to adapt the time-step to the speed of the simulation or even to the prototype's behaviour. It would increase the success rate of the Digital Twin and enhance its performance.
- *Security*: security protocols protect the Digital Twin from possible attacks. It is a superfluous problem here, while being relevant in a real application, especially if industrial or commercial. For a hypothetical lunar case, one might wonder how concrete the risk of hacking would be and which level of security would be convenient to implement. *Contact sensors integration*: here, sensors to assess the state of the prototype are considered; however, for more interesting results, it would useful to measure the impact of the docking interface. That would anyway add little complexity and lines of code, affecting the digital model more than the middleware. *Capability to access stored data of previous runs*: this point would strongly empower Digital Twin additional algorithms, such filtering, machine learning and AI, which use old data to increase performance. It would be important to help to identify faults, enhancing predictions and safety measures. However, also in the current project, old storage is not cancelled at the end of a run, being available for external analyses; simply, with the current simplified architecture, old data are not readable and employable by the Digital Twin.

5.2.2 Architecture of the Digital Twin

Having the functions clarified, it is possible to approach the definition of Digital Twin architecture. This step helps in its development, by highlighting connections and data flows. It is thus both a structural and temporal planning tool. It is important to start from functional considerations, in order to correctly consider inputs, outputs and relations.

The architecture is shown in figure 5.3. Such architecture is a high-level one, not yet a detailed structure of all the Digital Twin, which is possible to depict only in an advanced phase of the design. During the preliminary definition, any detail is liable of changes and reorganizations.

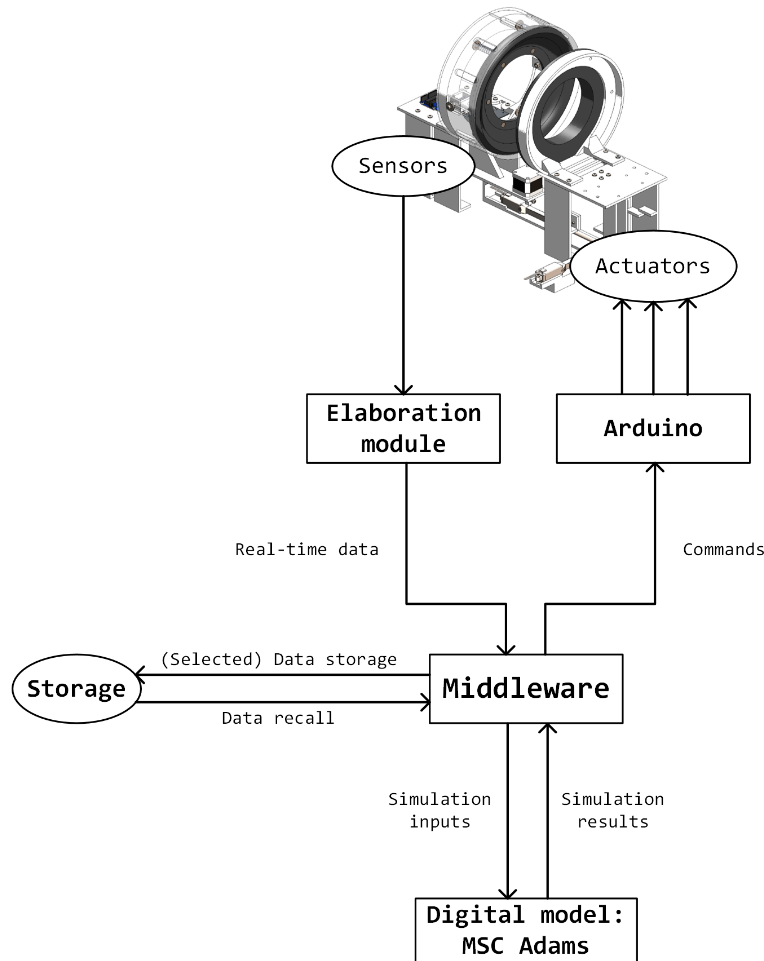

Figure 5.3: Architectural diagram of the Digital Twin. The *Elaboration module* may be considered both inside or outside the DT.

As from figure 5.3, the Digital Twin composes of three elements (and one 'bonus'):

- The *Middleware* is the core of the Digital Twin. Its role is to orchestrate data exchanges, request actions and manage timescales. Many Digital Twins use the middleware only for data flow management, while for the elaboration and the decision-making abilities use a separate block, the main one. For the purposes of this work, with limited complexity, it is preferable to combine all such functions, except for the simulation capability, into a single block.
  Being the main segment, the middleware is also the most complex, and its precise structure is defined later on (see section 6.3 for a description).

- The *Digital Model* is the simulation unity and is built on commercial software MSC Adams, as seen in section 5.1. It has to simulate the real system with the correct initial conditions and such simulation shall be initiated by the middleware.

- The *Storage* is the database collecting past states of the real system, as sensors data, commands and simulations results. It is primarily needed for taking decisions based on the history of the system, but may be exploited for advanced algorithms (such as machine learning and AI) and filters. The storage is located in the same computer of the other unities, being an internal memory.

- The *Elaboration module* is the 'bonus' unity. It indeed lays on a middle level between the middleware and the sensors. It however concerns a crucial architectural choice. As better seen in section 6.1, a convenient approach is to virtualize all kinds of incoming sensors, to have them available for creating possible other virtual sensors. The elaboration should elaboration should also include data filtering, to improve the quality.
  This type of processing may be partially done at the level of the sensors, before sending data to the Digital Twin; this, however, would lower flexibility and modularity. It is therefore suggested to include this elaboration unity inside the Digital Twin, to expand its control capabilities.
  This segment has been called 'bonus' because it *can* be included in the middleware, even though it has a separate role. Data filtering and elaboration are functions that are needed even without a Digital Twin (as in control systems), so they are not specific to it. However, the possibility to incorporate this module into the Digital Twin, makes it a component. Whether to merge it into the middleware itself is decided later, after preliminary analysis.

Data flow: from prototype to simulation

The Digital Twin is a real-time digital model with bidirectional communication, so it is worth dwelling on operations in the two directions. The first half of the flow is from the prototype to the digital model, as shown in figure 5.4.

The prototype, as real system, has ongoing physical processes which are sensed by some *sensor*. These sensors receive stimuli and convert them in digital signals, which can be sent to the Digital Twin's computer. Communication may both rely on wired or wireless technologies. Before of after data transmission, there may be the elaboration unity, here inserted after data communication. Sensors, data elaboration and communication strategies should receive a specific analysis, on a lower level plane; therefore, more it is said in section 6.1.
In this architecture, when data arrives at the Digital Twin, it enters the *elaboration module*. Its tasks are:

- If needed, to convert data in a format readable by the middleware.

- To filter data, cleaning it for a reliable reading.

- To possibly virtualize sensors.

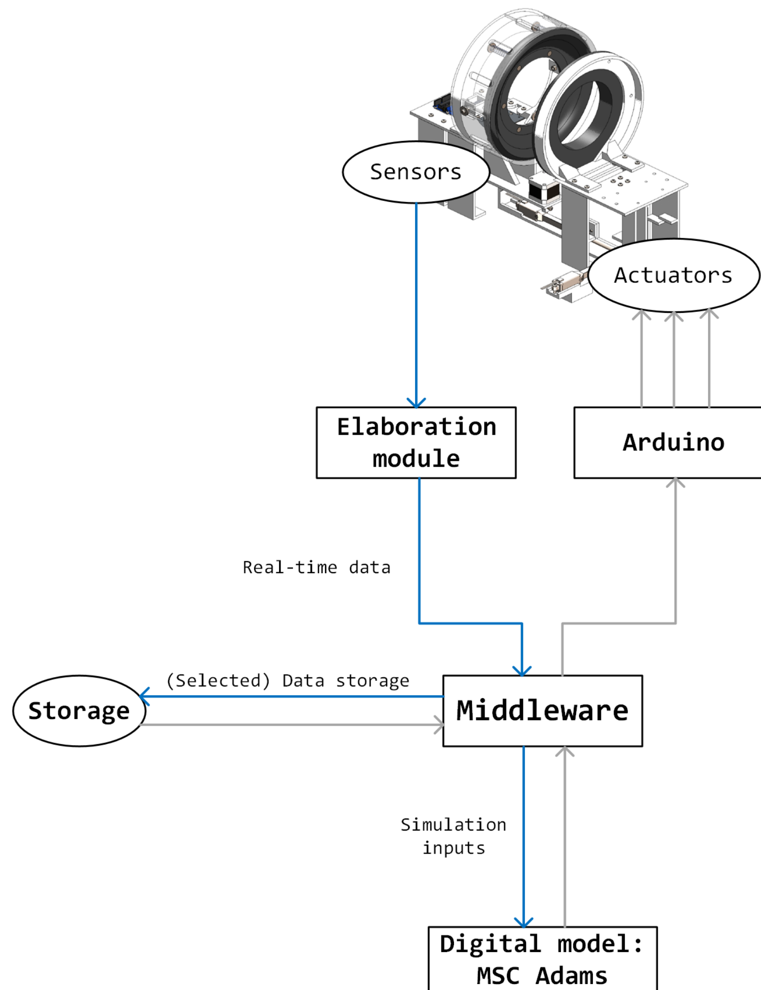

Figure 5.4: Command and data flow from the prototype to the digital model. To be read from top to bottom.

- To possibly combine virtualized sensors to build additional virtual sensors.

The communication from the *elaboration module* to the middleware sends almost real-time data (with natural delays due to reworking and data transferring). Depending on where the *elaboration module* is, such communication may be wired, wireless or inter-process. For this work, it will be inter-process communication, since the elaboration module is part of the middleware.

The *middleware* elaborates the incoming data, in the many ways needed: it may check the health status of the prototype and possible corrupt data. It is important to immediately check for possible faults, in order to stop the prototype as soon as possible. The main function of the middleware is still the orchestration of requests (sensors, storage, simulator), data exchange and commands. Such functions are time-dependent, being time the main variable, and include decisions. At the level of this work, decisions are simply

deterministic, without the implementation of more advanced algorithms.
One of the middleware's tasks is *storage* management: specific data describing the state of the real system are saved in the storage, for future availability.

The middleware then selects data for the simulation, inputs needed by the *digital model*. By sending the inputs and requesting the simulation, it asks Adams to run and predict the future state of the system. So, launching the simulation closes the first part of the loop.

Data flow: from simulation to prototype

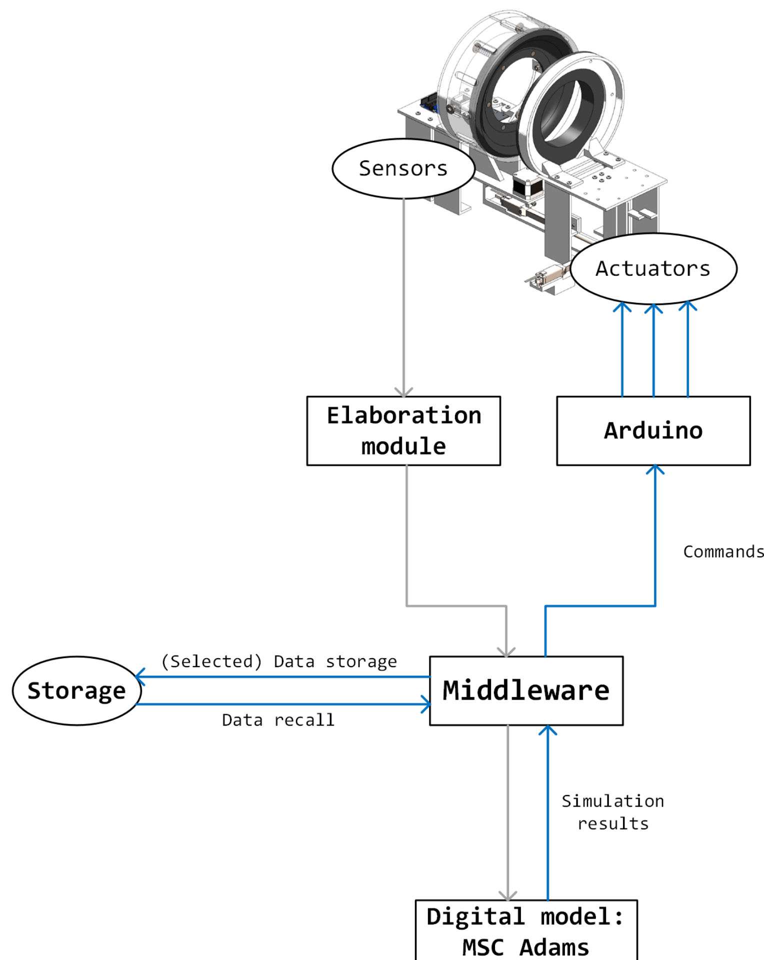

Figure 5.5: Commands and data flow from the digital model to the prototype. To be read from bottom to top.

Figure 5.5 depicts the way back of data, flowing from the digital model to the prototype. The stream begins with the *digital model* completing a simulation: it returns, as output, the predicted state of the system at the end of the time-step (which is still a near future

instant). Such data shall be read by the *middleware*.
In this phase, the *middleware* manages to:

- Receive simulation results from the digital model;
- Save simulated data in the *storage*;
- Possibly request and read old data from the storage; the middleware may conserve data from the last time-step to reduce inter-process exchanges;
- Compare data to take deterministic decisions about the prototype;
- Formalize such decisions, converting them in a format that can be sent to the prototype;
- Send commands to the prototype.

To resume, after the *digital model*-to-*middleware* communication, the middleware's interactions are so with the *storage*, for data saving and data recall, and with the prototype, by means of Arduino. The rest of the process is data handling in the *middleware*.

Further considerations

In the diagrams, there are three arrows connecting *Arduino* platform to the *actuators*. The reason is that the controlled degrees of freedom are managed with three actuators, each requiring a different command: two linear guides and one stepper motor, as in section 4.4.2.

What is missing in the diagrams is the interface with the human operator, naively included inside the middleware block.

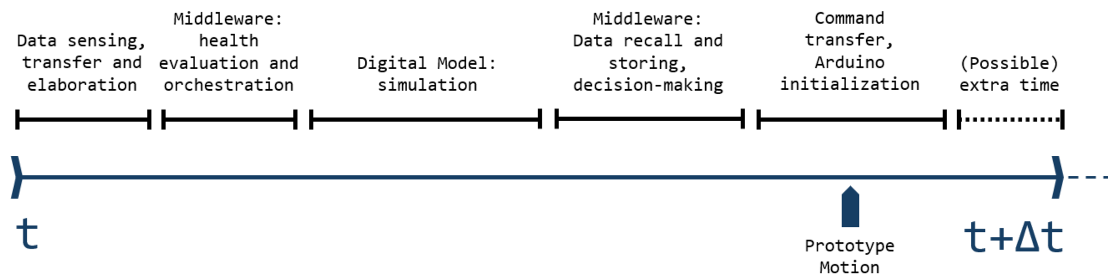

Figure 5.6: Logical temporal sequence of a single time-step.

This final space is reserved to discussing time advance in the Digital Twin. Figure 5.6 shows how a time interval is covered by the actions of the Digital Twin, following the round-trip loop described in the last paragraphs. Note that the proportions of the intervals in the picture are arbitrary and not necessarily coherent with the real length of each actions.

The process starts with sensing, data transfer to the Digital Twin and data filtering; this action may be initiated by a request of the middleware. The signal passes to the middleware, which evaluates the health status and starts the simulation.
After that, there is the simulation, which probably covers the majority of the time-step. The outputs are then sent to the middleware, which converts them in commands to be sent to the prototype and saves relevant information in the storage. Finally, the commands are executed; there may still be some extra time before the next time-step.

It is important to carefully manage such time-step; the reasons are:

- If the time-step is too short, there is not time for the process to finish before the end of the interval. The simulation mostly depends on the lightness of the digital model. Obviously, reducing the time-step Δt reduces the time-step to be simulated; the time needed by MSC Adams is linearly dependent on the time interval to be simulated, therefore it roughly involves a certain ratio of the CPU time-step. This ratio depends on the lightness of the model and on the hardware performance. The first necessity is to built a sustainable digital model.
  On the contrary, other processes that do have a fixed length, data storing, for example, depends on the amount of data to store, not on the length of the interval. The second need is to define a time step sufficiently wide to include also these fixed-duration tasks.
  If computer operations exceed the prefixed duration Δt, this would bring the Digital to crash. Early versions of the Digital Twin shall introduce the capability to remedy to such problem without crashes, while advanced upgrades would introduce the capability to adapt the time-step.

- If the time step is too long, there is abundant extra-time at the end of the operations. Such a problem would cause a loss of precision in the control, since a run would be split in few intervals.
  Moreover, the bigger this exceeding time is, the more the discrepancy between the predicted state and the actual one. The goal is to reduce the time between the actuation of commands and the beginning of a new time-step.

For these considerations, it will be essential to further analyse the temporal scale of the process during the calibration phase. Additional effort should improve the performance of the Digital Twin.

5.2.3 Software selection

Considerations on middleware

The architectural definition has shown how the core of the Digital Twin is the *middleware*. In this work, it has both mastering and decisional roles.
Literature highlights how pre-existing software may act as middleware, with some even specifically conceived for this. An example: Eclipse DITTO is an open-source platform helping to implement a Digital Twin, by exploiting JASON format for information exchange.[54] It is a middleware platform for IoT applications, especially for combining many different sources.

Also, service companies are developing their own commercial platforms, such as Cognite for heavy industries, or Oracle IoT Cloud from Oracle.

Given the academic nature of this work, it has been decided not to exploit such specific commercial tools. The choice remains between open-source specific platforms and custom software, based on languages such as Python or MATLAB (examples, respectively, of an open-source language and of a licence-based language).

Software choice

As said, the choice is between a specific platform or a custom one.
It must be considered that this is an academic and simplified project, not applicative; for this reason, it is reasonable to exclude a specific platform, such as Eclipse DITTO, which is valuable for practical Digital Twins with many inputs, outputs and a complex autonomy.

It has been chosen a custom-made middleware, built in its parts piece by piece. Such an approach gives maximal flexibility, however slowing the development.
The final decision was between open-source and commercial languages. Here, it was decided to exploit the university licence of MATLAB/Simulink. In particular, Simulink is a good option, advantageous in many phases of the work, as suggested by [5].

Simulink has indeed helpful interfaces that may be exploited during the project:

- Simulink/MATLAB combines with Adams Plant Controls, a tool that allows MSC Adams to simulate in combination with external tools, supplying inputs or using the outputs.
- Simulink is compatible with FMI (Functional Mockup Interface), also available for Adams files. Such property may be exploited; more is said in section 7.3.
- Simulink may act as a controller for Arduino with the package 'Support Package for Arduino'; it is not, again, a mandatory path, but it may help in the development of the Digital Twin.

Considerations on storage

For the storage, it has been decided to use a simple approach. The options are:

- *Excel file* (namely .xls or .xlsx), simple to read, but limited in terms of performance for writing and reading (considering MATLAB or Simulink);
- *Text file* (.txt), a small file format which is quite fast to be elaborated; it can give however problems in formatting complex data and may create problems in passing from strings to numbers;
- *JSON file*, easy to be read, both machine and human side, also supports complex structures of data.

Also here, it is not already taken a decision on the type of file used, but the JSON appears to be the best, and the .xls format the worst.

It should be noted that, for better storage management (both RAM and ROM), variables and models should be optimized and reduced to avoid heavy versions.[5]

Chapter 6

Digital Twin development

With the preliminary analysis completed, it is finally time to develop the Digital Twin. This chapter explores the design phase, being the heart of the work. Sensors are introduced and described, and so are the digital model and the middleware framework.

6.1 Review on sensing

Before entering the actual design process, it is useful to introduce sensors and their implications, with a section designated to this. The main goal is to pave the way to the choice of sensors, which is important for the overall project, before going into the Digital Twin.

6.1.1 Sensors: an introduction

Control loops

Here is introduced the concept of *control*, based on various information taken from [57].

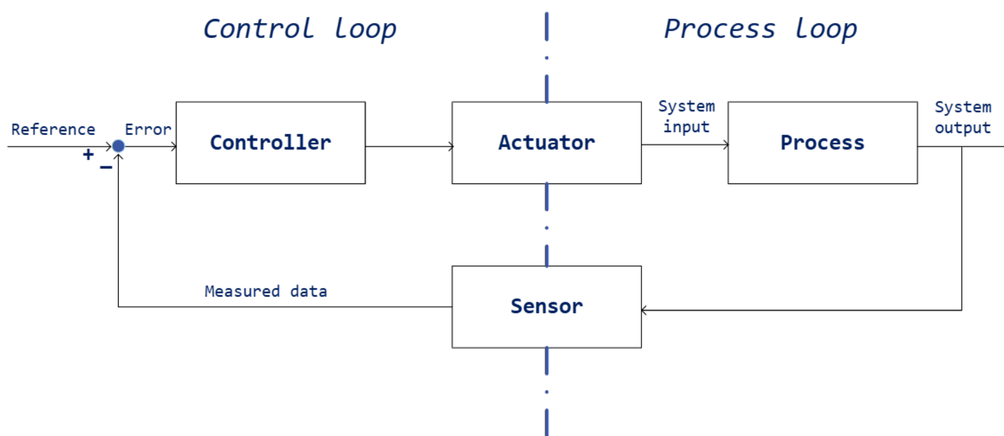

Figure 6.1: Scheme of a closed control loop. [57]

Control loop is the fundamental temporal and logical element of control systems. Control loops may be *closed-loops*, which rely on feedback principles, but also *open-loops* (not actual cycles) with a linear progress ending with the physical process. Here, the focus is on closed-loops.
A *closed control loop* is composed of two main portions, as in figure 6.1: the *control line* and the*process line*. Both occur in the same time.

- The *control line* is the engineered part. It begins with a sensor receiving information from the system and sending it to the controller, which generally compares data with the desired result. Being a closed loop, the *feedback* characterization is fundamental: the error resulting from this comparison induces the actuator to perform the required action for which it has been designed.

- The *process loop* is instead the physical one and, in a certain sense, the natural evolution of the system. It begins with the output action of the actuator, which is in fact the input for the system. It spontaneously evolves into a new state (even possibly without sensible changes), which is again measured by the sensor, permitting the cycle to begin again.

In general, it can be said that data undergo *information processing* between the sensor and the controller; moreover, actual control process usually gets more complex than this, with models, filters, decision-makers and virtual sensors.

The current work considers the Digital Twin acting as the controller in figure 6.1. It is therefore one of the simplest applications of Digital Twins.

Sensors

A *sensor* is a component, or a subsystem, installed to provide reliable measurements of the environment, to monitor processes and converting them into a signal readable by a human or by a machine. The purpose is to do it the highest accuracy, the lowest delay and minimal noise.
Here, a glimpse of control theory about sensors is displayed.

The *signal* $m(t)$ coming from the sensor can be described as $m(t) = h(y(t)) + \eta(t)$; $y(t)$ is the measured signal, $h()$ its conversion into the signal format, while $\eta(t)$ the undesired error associated to the measurement. Since $h()$ generally depends on some internal variable of the sensor (typical of the functioning of the sensor itself, not depending on the sensed system), called $z(t)$, the sensor assumes a generic dynamic system form:

$$\frac{d}{dt}z(t) = f(z(t), y(t))$$

$$m(t) = h(z(t), y(t)) + \eta(t)$$

Therefore, the signal depends not only on the system, but also on this internal variable $z(t)$, the evolution of which is indicated by the first equation. The signal $m(t)$ may undergo to a first filtering (or linearization, or extrapolation,...) by the sensor's software, providing a final signal $m'(t)$. Not to be confused with the derivative of $m(t)$, $m'(t)$ needs to be

the most representative of the measured phenomenon as possible. Namely, the goal is $m'(t) \simeq y(t)$, which means reducing the effects of measurement deformation and noise; it is consequent that minimizing the error is desired, $\eta(t) \to 0$.

Furthermore, a *linear* or *linearized sensor* may be defined, with coefficients in the following form.

$$\frac{d}{dt}z(t) = A_s z(t) + b_s y(t)$$

$$m(t) = c_s z(t) + \eta(t)$$

With this simplification, the signal linearly depends only on *z(t)*, the sensor's internal variable. Linearization may hold just in limited ranges of functioning and, but in that specific neighbourhood helps to simplify the operating law.

Moving on to a slightly general perspective, it is now shown the dynamic model of a *plant*, which is the system to control in *control theory.* Assume: an *input vector u(t)*, an *output disturbance vector v(t)*, an *input disturbance vector w(t)*, the *state vector x(t)*, the *output vector y(t)*, and the *required information z(t)*, note that *z(t)* has a different meaning than before. These are vectors rather that single numbers, since many actors may appear in the phenomena; thus it holds $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$, $w \in \mathbb{R}^s$, $y \in \mathbb{R}^p$, $z \in \mathbb{R}^q$, $v \in \mathbb{R}^r$. The plant can be assumed as a linear model:

$$\frac{d}{dt}x(t) = Ax(t) + B_1 u(t) + B_2 w(t), \quad x(0) = x_0$$

$$y(t) = Cx(t) + v(t)$$

$$z(t) = Dx(t)$$

The first equation means that the state vector evolves taking into account its own state, possible inputs of the system *(u(t))* and possible disturbances *(w(t))*; the initial condition is defined. The second equation evaluates that the output *(y(t))* depends on the state vector *x(t)* by a *linearized* relation, with the output disturbance *v(t)* that introduces perturbations. The third equation gives information about the required information *z(t)*, again linearized.

Introducing virtual sensors

Often, in control problems, the number of required variables is higher then the number of measured variables: since some are dependent on others and can be obtained from others, not each unknown must be directly sensed. The practice of computing variables from others is the most trivial example of a *virtual sensor*, concept introduced in the next pages.

Thus, referring to the last equations for the linear sensor model, an observation can be made on the size of the vectors: $n \geq q > p$. The inequality means that the number of required variables q (dimension of z) is not bigger than the number of state-defining variables (n, dimension of *x(t)*); the dimension of the output, p, is smaller.

This last consideration is not a universal law, but means that not every variable is measured ($q > p$), for practicality (as later detailed). Indeed, the output *v(t)* is arbitrary and generally stands for the measured variables. By combining the two inequalities, it

follows that $n > p$, meaning that the number of measures is smaller than the dimension of the system's state vector.

Instead of directly measure, a *virtual sensor* elaborates the signal of various real sensors $z_i(t)$ (or even other virtual sensors) to estimate a different variable. As a sensor is intended to give *z(t)*, the virtual sensor has its $\hat{z}(t)$, allowing to define the *sensing error* $\tilde{z}(t) = z(t) - \hat{z}(t)$. *z(t)* would indeed be the required variable if directly measured (net of perturbations) instead of being obtained by processing other variables, which happens to be the case of the virtual sensor $\hat{z}(t)$. However, since a virtual sensor generally replaces a non-existing sensor, *z(t)* is not available, and so is the error.
Another consideration is that the error $\tilde{z}(t)$ is preferable to have low sensitivity to initial conditions, to time t, to the disturbance of the real system and to the noise of the effective measurements; this gives stability, making the errors more predictable and manageable.

6.1.2 Virtual sensors

Sensors are fundamental for system monitoring and so are for Digital Twins, which rely on information from the real body. As already seen, physical sensors, both analogic and digital, convert the measurement of a phenomenon into a signal of a different type, readable by an observer (human or machine). A classical thermometer is a sensor: it gives a visual output to read the temperature value. Even our five senses are classical sensors, for example transforming temperature and heat inputs in a nervous impulse.
However, physical sensors show some weaknesses:

- Sometimes sensors are impossible to be placed in the exact point where they are needed, because of obstructed space or harsh conditions.
- Sensors are quite expensive and require power, other reasons for which it is generally not possible to globally cover an environment. In general, software is less expensive than hardware, mostly for cutting-edge applications; this is even more true when updates are needed.
- Often sensors do not provide continuous signals, but discrete values on a periodic scale. Thus, real sensors architectures both discretize the temporal scale, both the spatial one.
- A single sensor is exposed to faults, temporary unavailabilities and performance decline (sensor *drift*) during its life-cycle.
- *Noise* is an innate undesired consequence of sensing, and combining more sensors may reduce its impact.
- Sensors are not always replaceable, such as in many Space applications.
- Sensors have an intrinsic delay in detecting, elaborating and transmitting the signal to the controller.

Virtual sensing represents a solution to some of these limitations. It does not consist of direct measurements, but in the combination of pre-existing data, such that arising from

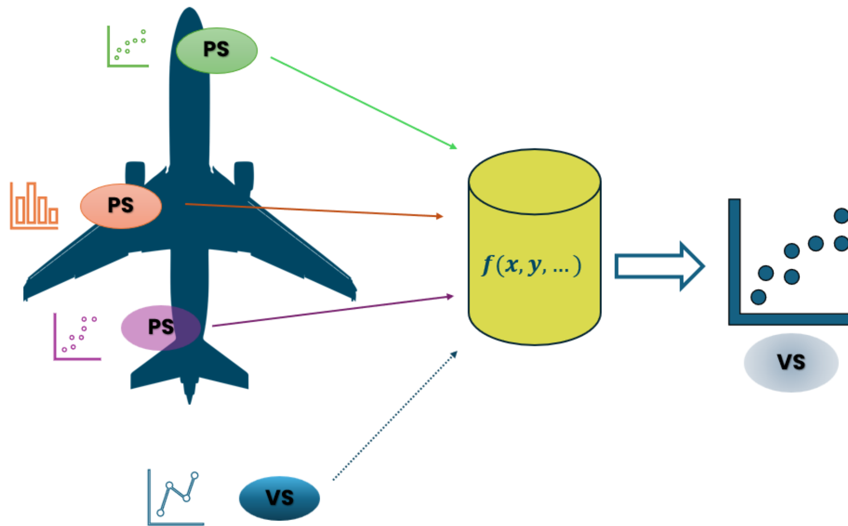

Figure 6.2: Schematic representation of a virtual sensor, based on [58]. At left, a choice of inputs is shown, including physical sensors on the real asset and a virtual sensors; in the centre is the data fusion function which returns data, as seen on the right as output.

different physical sensors.[59] Virtual sensors are also called soft sensors, smart sensors or estimators.
The idea is to process available signals from physical sensors, even some measuring different phenomena, with predictive algorithms that can provide an output, simulating a sensor. Input data may derive from one or more physical sensors, but also one or more virtual sensors, or even a mix of them, as in figure 6.2; at the very source, data obviously derive from physical sensors.

This technology offers, as one of the main advantages, the possibility to simulate sensors in lacking spots, elaborating information acquired at other locations; analogously, it may help to tighten the temporal mesh of data. Moreover, by updating the software, virtual sensors can be redesigned during the operative life.[58] However, one critical point is the necessary trade-off between data quality and its weight on computations, storage and transfer. Moreover, virtual sensors are not magical; if fundamental data are not available, some variables may not be extracted: there are cases in which additional hardware is the only possible solution.
Examples of virtual sensors arise from many fields, from aerospace and chemistry to computer science and acoustic. This approach is fundamental in the implementation of IoT technologies.

Martin et al. propose the following classification of virtual sensors [58]:

- *Sensor virtualization*: the signal from a single physical sensor is mirrored, possibly filtered or cleaned, and may undergo processing (e.g. acceleration data may be converted into steps by a pedometer). However, it is not yet combined with signals from other sensors. It might be observed that also real sensors use physical

correlations to describe a phenomenon and to convert it into an electric signal, generally also including a software component: they are in some sense virtual sensors themselves.

- *Competitive sensing*: different sensors measuring the same property are combined into a single signal. It is a redundancy strategy to prevent faults and noise consequences to improve reliability.
- *Static cooperative sensing*: the elaboration of data coming from different types of sensors allows to extract different data, not directly measured.
- *Dynamic cooperative sensing*: as for the previous type, distinct sensors are combined to obtain new information; the difference lies in the architecture, since input set is dynamically changing and may include different inputs at different times.

Another useful ability for a Digital Twin is to handle, rearrange and mix sensors data to create further virtual sensors with latest software updates; such an approach would add flexibility and modularity. Thus, even if it adds a level of complexity and latency, it may be helpful to virtualize each physical sensor for the most flexible architecture possible. It is not a new concept; for example, it was already proposed and formalized, among others, by Madoka Yuriyama and Takayuki Kushida in [60].

It is important, when starting to design a virtual sensor, to specify at least the following points: [61]

- *Input* data: it is important to define the types of data required, including the sufficient and necessary data needed, as well as identifying possible redundancies and other heterogeneous data sources than can enhance the quality. For an adaptable framework, it may be considered that inputs are not static and may change in time, with less or more available signals; the virtual sensor must however be able to provide the same output, at most clarifying a variation in accuracy.
- *Aggregator*: the actual function elaborating the inputs to calculate the wanted variable (or variables); filters should be included;
- Resulting *output* data: the variable type returned by the virtual sensor.
- *Aggregation frequency*: the frequency of each output computation, which may be non-uniform with the sampling rate of the inputs. Interpolations and extrapolations may be considered, with a proper rationale. For example, most recent data should have more weight than older data.

Also metadata describing the produced data packages would require a specific analysis.

Architectural considerations

[61] contains a proposal for an application allowing users to build and use virtual sensors on the base of physical ones. The architecture proposed by the authors distinguishes between continuous data requests from the sensors and one-time-only queries. To contextualize,

for a Digital Twin the request should be continuous for almost all cases, entering the field of sampling rates; however, impact measurements (such as those taken in a docking event) may be an example of non-continuous data exchange.
It is useful to examine other architectural considerations emerging from the article. In particular, virtual sensors might have a reserved accommodation in the architecture (of the application, in the article; of the Digital Twin, in this work), in order to be independent and accessible, similarly to a physical sensor.
In the same regard, an hardware perspective is also evaluated. Virtual sensors are digital, but need a physical place for data processing and storage. In articulate systems, these may be located in many places:

- Inside a frequently used real sensor, trying to reduce the overall exchange of data. However, it is not a useful strategy if also raw data from real sensor has to be sent to the main computer (for instance, the middleware of the Digital Twin), only resulting in more data to be transmitted. Such a solution is convenient, instead, if one or more physical sensors are needed only for this virtual measurement, which would be the only data package to be transferred to the main computer. Useful cases can be *sensor virtualization* and *cooperative sensing*, in which the virtualization process output is not bigger than the input.

- Virtual sensors can be elaborated in the principal operational unit, or in the middleware unit (if the two are separate). This solution collects inputs and centralises data elaboration.

- In an apposite external unit, both detached from real sensors and the main computer, if required by environmental reasons.

- Data elaboration and storage (specific to virtual sensors) may be located in different places and redundant. Such considerations concerns both the hardware both the software architectures.

6.1.3 Middleware and virtual sensors

A seen, a *middleware* is generally introduced for communication with sensors and virtual sensors and, in general, in Digital Twins. It is a module dedicated to communication and, often, storage.

Indeed, for communication feasibility, *device abstraction* is useful, if not necessary: all components, such as sensors, need to be levelled out to make them resemble as uniform units, regardless of the assorted communication protocols possible. It means that an intermediate wrapper, that can be the middleware, has to translate between the principal language to the ones of the single different devices. It hides, for and high-level observer, the differences in hardware and languages, linking sensors to the main framework.
In this specific work, the middleware acts also as orchestrator of the whole system, as seen in section 5.2.

Focusing on the management of (virtual) sensors networks, progress has been made over the years. A modular and flexible framework was already proposed with the open-source

Global Sensor Networks (GSN) project in 2005.[62] Developed at EPFL in Lausanne with Java, C and C++, GSN is a middleware devised to simplify the creation of applications based on these technologies. Developers started from abstract requirements of virtual sensors networks and built an architecture capable of easily integrate new sensors, in an adaptive and light environment. Such a sensor deployment requires XML and Java knowledge, being not trivial for beginner users. One of the focal points of the project was the time management of data packages along their path, mostly inside the query mechanism. See [62] for details about the architecture of the Global Sensor Networks, with a functional description of its layers.

In the following years, with the spread of IoT concepts, many alternatives have been created also outside academia, while GSN remained more of an architectural proposal than a piece of software that was actually used. Many examples up till 2015 are collected in a IoT survey, which catalogues middleware options in categories.[63] Ten year having passed, the article has now more relevance for gathering features and requirements for such platforms, rather than representing a useful catalogue. Some of the common features identified are:

- to deal with heterogeneous devices;
- to comply with limited and variable resources for sensors and data exchange;
- to handle large amount of data;
- to run automatically and with spontaneous interactions with sensors;
- to be context-aware, possibly using informed models;
- to be real-time.

Clearly, these concepts are well-fitting for IoT environments, but would need a slight reshaping if moved to a Digital Twin application. Speaking of the identified requirements, some are:

- resource management and discovery, since sources can vary in availability and heterogeneity;
- data management;
- events management;
- timeliness;
- continuous availability;
- interoperability;
- autonomy.

Please refer to [63] for a detailed analysis.

Not all these features are necessary for a simple framework, as the one of this work, but they are a basis for more extensive projects, especially when managing a multiplicity of sensors and devices.

It should be said that many examples of middleware are found on the market, and many are more recent than [63]; IoT tools may also be adopted. Examples are Eclipse Hono, ThingsBoard, Mainflux, Thingspeak, Ptolemy Accessor Host. For advanced applications, it may be decided to use one. As seen in section 5.2.3, however, it was decided to use MATLAB/Simulink for all the functions, building a custom middleware (also acting as central unity).

6.2 Sensors selection

Sensors and virtualization have been covered, to suggest possible software approaches. However, as seen in section 4.4.2, the prototype does not yet have sensors. This section focuses on possible options for integrating it with one or more sensors, essential to implement a control system and, above all, a Digital Twin.

6.2.1 Sensors options for the docking prototype

The prototype needs sensors; however, many physical entities may be measured, leaving open the question on which to measure and how. Table 6.1 lists options, both for the properties to be measured, both for the sensors to employ.
It is however a preliminary list, since many of the options are not of practical use for the current configuration. Reasons may be:

- Limited budget;
- Size constraints: prototype parts have centimetres size and not all sensors have suitable dimensions;
- Mass constraints: there are sensors that might be mounted on the prototype but have sizes comparable with the ones of the part itself. This is a problem, since the sensor mass gets relevant and adulterates prototype's values. Therefore, sensors should be small enough. Such a reasoning may be by-passed if sensors integration is considered from the beginning of the design loop. If the sensor has a relevant mass, mock-up counterweights may be added to keep inertial symmetries.
- Sensor uncertainty shall be coherent with the orders of magnitudes of measured quantities and with the environmental set-up. Sensitivity must be good enough to detect variable variations, but neither being too precise, otherwise polluting the measurement with environmental noise.
- The sensor shall not interfere with the magnets of the docking system.
- The sensor shall operate in the g-range typical of the impact, which must be preliminary estimated.

The best options, filtered for the relevance of the measured quantity, shall be further analysed in a trade-off, which is in section 6.2.2. The role is, as said, to provide a feedback for the Digital Twin.

What to measure	Unity of measure	Sensors options
Linear misalignments	m	Linear encoder; Radio sensor; Laser sensor; Ultrasound sensor; Optical sensor
Angular misalignments	^{o}	Gyroscope; Incremental encoder; Optical sensor; Magnetic sensor; Absolute encoder; Laser sensor; Radio-finder; Optical sensor
Distances	m	Linear encoder; Radio-finder; Active or passive triangulation; Laser sensor; LVDT (Linear Variable Differential Transformer); Optical sensor
Relative velocities	m/s	Linear encoder; LVDT, Accelerometer; Optical sensor
Accelerations	m/s^2	Accelerometer
Forces	N or m/s^2	Force sensors; Accelerometer; LVDT; Laser sensor
Actuators parameters	Various units	Current/Voltage sensor; Force sensor; Torque Measurement; Measurement of the effects (see row above)

Table 6.1: Naive miscellany of possible sensors to be integrated with the prototype.

Sensors for impacts

The primary purpose of sensors, for the current work, is to supply inputs to the Digital Twin. However, a possible secondary mission may be to investigate how the prototype behaviour during the docking impact. It is clearly not necessary for building the Digital Twin, while being the first applicative and useful investigation about the behaviour of the prototype.

For this last reason, here it is a short preliminary analysis of sensors for collision sensing. The object of the measurement is mainly the exchanged force between the chaser and the target, in order to characterize the stress experienced the rover and the habitat (with the necessary observations of adopting a small scale prototype and immature design).

- *Strain gauge*: used for strain measurement; a multiple strain gauges architecture may be implemented to detect strain on different directions.
  These sensors are more appropriate for detecting internal deformation, less for global

impacts. Also, many factors discourage strain sensing: the annular shape of the docking interfaces, with many small-sized holes, does not offer much free surface; the 3D-printed material is not isotropic, and many measures or a complex architecture would be needed for accurate strain mapping; shocks are strong but not extreme, so internal material response is not the first need.

- *Load cell*: a force transducer, it reports dynamics quantities. Load cells are generally too big for the prototype and give the best results in static or quasi-static cases.

- *Accelerometer* is probably the best options. Extremely various in type, size and price, there probably exist a solution representing the best compromise for each situation (at least at this level of complexity). It would indeed be sufficient to measure accelerations on the female interface to have a characterization of the impact.

- Advanced technologies are perhaps too precise for this application and in certain cases not applicable to the prototype configuration. Environmental contaminations and test bench limitations are not low enough to not pollute subtle measures with noise. Examples of such sensors might be piezoelectric crystals or laser systems.

An 'unfair' option: simulated sensors

As this work is a simple technological demonstration, a peculiar alternative may be explored. Being in the development phase, indeed, sensors are needed to build the Digital Twin but not yet for precise testing: it might be proposed to simulate these sensors.
This possibility can be considered for the development of the Digital Twin, but should later being replaced by real sensors for the precise control of the prototype. Here, the purpose is just to build the control system and the Digital Twin, with its architecture, its protocols and its interfaces.
Although seeming a simple option, some considerations must not be forgotten:

- Being integrated in a control loop, such a solution may be implemented only for simple phenomena, which can be forecast with reliability. A *model* must be developed to replace the sensor prediction, therefore it should give good results and not being more complex and onerous than installing an actual sensor. However, all considered, it is inevitable that an error, even minimal, will accumulate between the simulation (fake control loop) and the actual prototype; this is acceptable only for short runs, that leave no time for the integral error to diverge, and for preliminary test phases.

- The *model* needs inputs to compute the outputs to send to the Digital Twin. Therefore, an additional link should be added to the classical Digital Twin architecture (seen in figure 5.3): the Twin sends some sort of data to the simulated sensor module, which elaborates it into a signal, giving birth to the flow in figure 5.4.

- The simulated sensor needs a validation. Meticulous or not, it depends on how early it will replaced by the real sensor: more accurate validation may be required if the simulated sensor is employed up to the end of the design phase, having to ensure

high fidelity in the 'measurements'. The minimum level of validation ensures that the simulated sensor does not indirectly damage the hardware.

- The simulated sensor shall be integrated in the architecture paying attention to the ease of being replaced by the real sensor after the development. It must be indeed a tool for the project and not part of the final system, which requires a real feedback.

- The model needs to correctly predict evolution of physical quantities, but also has to insert *noise*, coherent to the one of a sensor. It is mainly used to test the response of the Digital Twin to measurement noise.

- Similarly, *faults* should be simulated, in order to evaluate malfunctionings.

- The simulated sensor shall be created with the lowest trace of its model, trying to be indistinguishable from a real sensor, being a black box at the eyes of the Digital Twin. Else, the framework would not be ready to accommodate a real sensor.

- A simulated sensor is not a *virtual sensor*. The second one uses actual data and combines it with a model: its aim is to have the best estimate possible of otherwise absent data. A simulated sensor is a temporary tool that imitates a sensor in its behaviour: the focus is on its role inside the software/hardware architecture and not on data.

- Simulated sensors could also be used to design and test data transfer methods and protocols, if they were placed in the location of the real sensors, rather than inside the main computer unit.

6.2.2 Trade-off

It finally is time to choose the sensors to implement.
For a prototype project, this step may be taken after the completion of the mock-up, as in this case. Differently, for a real project, sensors design must be included in the design loop and considered in all its phases, with a better planning. The second case is general the better option, ensuring better performance.
Here, as said the situation is however the first, with the prototype ready and the sensors needed mostly to test the Digital Twin, rather then the prototype itself.

The first action is to individuate all the possible options, possibly removing the clearly unfeasible ones. A first, rough, list is in previous pages (see table 6.1). After this, a *trade-off* considers all the remaining options and evaluates them based on specific figures of merit, each having a specific weight in the decision.

Options

The purpose of this sensor, or sensors, is to evaluate the state vector of the chaser, relative to the target. In particular, it shall be considered that the target structure is fixed, while the target interface (the female part with magnets, undergoing the impact) is partially

free to move, as allowed by the springs.
The needed information are:

- The *position* of the target with respect to the stationary female interface (the support). Being planarly restricted, only the two horizontal components are needed, resulting in a 2D problem. Such coordinates would then be converted by the Digital Twin into linear distance, Delta and Gamma (as in section 4.4.2).

- The *velocity* of the chaser with respect to the fixed target's structure. It is sufficient just the component along the linear guide, having all the other components constrained. Other components would be associated to vibrations or to (mis)alignment operations, acting on the other actuators (Linear Guide 2 and Stepper motor; again refer to section 4.4.2).

A second, refined, list of sensors is then formulated:

- *Gyroscope*: a MEMS (Micro-Electro-Mechanical Systems) gyroscope may measure, indirectly, the pose between the target structure and the chaser. It is instead not suitable for distance or velocity.

- *Accelerometer*: a MEMS accelerometer mounted on the chaser may give its velocity and position along the linear guide. However, being derived measures (integrated from the acceleration), it is necessary to initialize the values; to avoid error accumulation, it is preferred at the beginning of each run.

- *IMU (Inertial Measurement Unit)*: it incorporates a gyroscope and an accelerometer, often with magnetometers, being just the union of the latters. Again, it would be needed a MEMS version.

- *Optical tracker*: to be put under or on the side of the chaser, measuring the path travelled. It could work with markers to understand the point the chaser is on its rail, or with more complex pattern recognitions to understand the relative distance of chaser and target interfaces. It may give linear position on the rail and, deriving, chaser's velocity. Integration complexity is higher then other physical-based sensors.

- *Linear encoder*: a good solution to measure linear distance and to obtain velocity; it however needs space for implementation of the tape, being invasive. It both may be *absolute* or *incremental*.

- *Angular magnetic encoder*: it is an option to measure the angular pose of the chaser with respect to the target if mounted on the shaft of the Stepper motor (angular actuator for misalignment γ). However, the presence of the docking magnets may be a source of disturbance and needs further verification and possible shielding. Moreover, it does not cover the measurement of Δ.

- *LVDT (Linear Variable Differential Transformer)*: although being very precise, it is bulky and indicated for very small distances, less the ones in the prototype.

- *Magnetic sensors for velocity*: each sensor detects the transit of a magnetic source (mounted on the moving body), allowing many aligned sensors to measure motion. It is a cheap option but requires many sensors; one considerable problem could be the interaction with the docking magnets.
- *Laser time-of-flight sensor*: measuring the time a laser beam takes for going to an object and coming back, they are able to compute the distance of the object and to extract its relative velocity. The field of view is however limited to $15^o - 20^o$. Two sensors may offer a 3-D estimation of the state vector, similar to two eyes.
- *Simulated sensor*: the last option has been explored in section 6.2.1 and is suitable just for the design and testing of the Digital Twin, not for an operative application in which real data would be needed. For this work it can be sufficient, thus it is taken into account.

Such options have been again considered with a simplified approach, this time in two categories:

- Sensors measuring linear distance (Linear Guide 1) and velocity
  - Accelerometer
  - IMU
  - Optical tracker
  - Linear encoder
  - Magnetic sensors
  - Laser time-of-flight sensors
  - Simulated sensor
  - *LVDT*: this possibility has been discarded, because not much suitable, as seen above.
- Sensors for measuring relative pose between chaser and target
  - Gyroscope
  - IMU
  - Angular magnetic encoder (measuring γ on the Stepper Motor) + linear encoder (measuring Δ on Linear Guide 2).
  - Laser time-of-flight sensor (triangulation with three sensors, or at least two)
  - Simulated sensor

These options have been organised in a table, in figure 6.3. Such a table considers on the columns the sensors for linear distance and velocity, and on the rows sensors for pose measurement.
The combination of these options gives many possible configurations. Some have been chosen, shown in green and numbered for clarity. The other combinations are redundant or even meaningless for the prototype: they are shown in red and not considered.

		Linear distance and velocity						
		Accelerometer	IMU	Optical tracker	Linear encoder	Magnetic sensor for velocity	Laser time-of-flight sensor	Simulated sensor
Relative pose	*Gyroscope*	1		2	3	4		
	IMU		5					
	Angular magnetic encoder + linear encoder	6		7	8	9		
	Laser time-of-flight sensor						10	
	Simulated sensor							11

Figure 6.3: Configuration matrix of the sensor options.

Some observations about the table can be made:

- Both *IMU* and the combination of multiple *laser time-of-flight sensors* are sufficient by themselves in measuring the state vector (relative position between chaser and target and chaser velocity), so are considered as two independent solutions and not in addition to other sensors, even if possible for refined measurements.
- *LVDT* has been discarded, as already mentioned some paragraph above.
- *Simulated sensor* is not of interested if combined with other sensor. It would indeed replace only pose detection or motion detection, while the advantage of simulated sensor is not to integrate real sensors (see filters otherwise). Therefore it was considered alone in a single configuration.

All options are ready for trade-off. However, eleven configurations is still a significant number. To reduce it, an arbitrary skimming has been made. The remaining configurations are shown here, along with the rationale:

- *Configuration 1*: gyroscope and accelerometer together may represent a custom-made IMU;
- *Configuration 5*: IMU, as already built, is considered as an option;
- *Configurations 7, 8, 9* have been kept; they measure the state vector with three sensors: one for angular misalignment γ, one for translational misalignment Δ and

one for the linear position and velocity.
The three configurations may be gathered in a unique case named *Configuration 789*, in which γ is measured on the stepper motor shaft by the angular magnetic encoder, Δ is measured by a linear encoder and linear position and velocity (along Linear Guide 1) are measured with the other sensor (optical tracker,a second linear encoder or a magnetic sensor). This option should undergo another trade-off to select the best of the three alternatives.

- *Configuration 10* considers a triangulation (or at least a bilateration, such as human eyes and 3D vision) with Laser time-of-flight sensors.

- *Configuration 11* is the last option, with a simulated sensor for all the variables.

Figures of merit

For the trade-off, the following figures of merit have been defined:

- *Cost*: sensors collocate in a wide span of prices; for this prototyping work, lower costing options are preferable and have higher score.

- *Implementation ease*: implementation considers the practical process of mounting on the prototype, considering that some solutions may require the addition of rails or other components. Easiest-to-mount and less-impacting configurations score higher.

- *Noise*: lower noise solutions are preferable and score higher. For the simulated sensor, the noise is considered as quite high; this choice is not due to the actual applied noise, whose magnitude can be imposed as wished, but because data precision is low. Indeed, the simulative model gradually diverges from the real case, by accumulating errors, and without taking responsibility for any faults. In fact, this figure of merit accounts for data quality.

- *Availability*: it is preferable that sensors are commercially easy to find, and such configurations score higher; this was also due to the time frame for a thesis.

- *Magnetic interaction*: neodymium magnets might interact with magnetic-based sensors, requiring additional analysis and possible shielding. Configurations without magnetic sensors score higher.

- *Versatility*: Sensors that can be used for further measurements, beyond assessing the state vector, score higher.

- *Calibration complexity*: More sensors require more validation and calibration. Candidates requiring less work on validation score higher. Simulated sensors needs validation and calibration.

- *Software complexity*: software is required for simulated sensor but also for elaborating data; for example, velocity can be derived from position, but it requires software lines to do it. The less software to write, the higher the score for this figure of merit.

It is important to notice that, for the the trade-off assessment, configurations consider average sensors, since there exist various precision and various price solutions for each class of sensor.

The figures of merit are weighted by one-to-one comparisons, which combined give the *prioritization matrix* is obtained. Figure 6.4 shows the results. The table shows the

Prioritization Matrix	Weight		Position
Cost	0.129	12.9%	4
Implementation ease	0.078	7.8%	5
Noise	0.274	27.4%	1
Availability	0.229	22.9%	2
Magnetic interaction	0.210	21.0%	3
Versatility	0.018	1.8%	8
Calibration complexity	0.037	3.7%	6
Software complexity	0.025	2.5%	7

Figure 6.4: Prioritization matrix with the weight of each figure of merit. Higher score means higher relevance in the trade-off; only outputs of the comparison matrices are shown, with ranking on the right.

resulting weight of each figure of merit; clearly, the sum of all the weights is one. Therefore, weights can be expressed as percentages, as in the central column. Finally, in the right column there is the ranking, according to relevance.
The most important figures of merit are *Noise*, which as said represents data quality, *Availability*, also due to the short the tight deadlines of the thesis, and *Magnetic interaction*.

Final choice

Defined the figures of merit and their weights in the trade-off, it is time to perform the trade-off among the various choices. Please remind that the candidate configurations are:

- *Accelerometer and gyroscope*, the former *configuration 1* in figure 6.3; it is considerable as a custom IMU.
- *IMU*, also *configuration 5*.
- *Encoders architecture*, also named *configuration 789*; as said, it is the combination of three configurations (*7, 8* and *9*): the best of the three is destined to be be chosen with a further analysis, in case of winning.

- *Laser time-of-flight sensors*, also *configuration 10.*
- *Simulated sensor*, or *configuration 11*, described in section 6.2.1.

The trade-off takes one-to-one comparisons between each couple of candidates and for each figure of merit. The results are reunited in a high-level matrix, which combines them with the weights of the figures of merit, giving the overall score of each configuration. The halfway and overall results are in figure 6.5.

Trade-off results	Cost	Implementation ease	Noise	Availability	Magnetic interaction	Versatility	Calibration complexity	Software complexity	**Total score**	
FOM weight	0,129	0,078	0,274	0,229	0,21	0,018	0,037	0,025		
Accelerometer and Gyroscope *(conf. 1)*	0,236	0,19	0,088	0,178	0,171	0,344	0,129	0,2	0,162	16%
IMU *(conf.5)*	0,158	0,246	0,171	0,111	0,08	0,344	0,115	0,2	0,144	14%
Encoders *(conf.789)*	0,071	0,04	0,349	0,065	0,037	0,107	0,442	0,502	0,161	16%
Laser time-of-flight *(conf.10)*	0,033	0,038	0,349	0,065	0,356	0,14	0,187	0,061	0,203	20%
Simulated sensor *(conf.11)*	0,502	0,486	0,043	0,581	0,356	0,064	0,126	0,037	0,329	33%

Figure 6.5: Results of the trade-off for the sensors.

The first row recalls the weights of each figure of merit, already seen in figure 6.4. All the first eight columns show how each candidate configuration scores for each figure of merit. These results are numbers between zero and one, expressible also as percentages; note that their sum (column by column) is the unit. The option obtaining more firsts is simulated sensor, but the results are quite balanced.
Combining such scores with the weights of figure of merits with a weighted sum, the final results are obtained and shown in the last right column, both as value and as percentage. The resulting ranking is:

1. The *simulated sensor* scores highest, with 33% of the preferences. It is thus the implemented choice; see the next chapters for the details.
2. The *laser time-of-flight* sensor has 20% of preferences and is kept as back-up option. It is therefore the main candidate for a real implementation after the development of the Digital Twin, when the simulated sensor would no longer be sufficient for actual control.
3. Both *accelerometer and gyroscope* and *encoders* architectures receive 16% of preferences. Even if distinct, the two options receive almost the same score. To be exact, the custom IMU has a slightly higher score.

4. The lowest performing option is *IMU*, with 14%. It is a bit less preferred with respect to the custom made option because it performs worse in terms of cost and has presence of magnetic elements. However, the last three options all score similarly, without a clear preference.

A last marginal note: before a physical sensor is implemented, it should be performed an analysis on the transmission line between the sensor and the Digital Twin computer. Data communication can be simply achieved using Arduino, but also with an external wired line or even wireless. These last options require a further communication interface, on the software side, with the Digital Twin. Moreover, the wireless option require bandwidth budget analysis, with possible supplementary requirements.
A more meticulous analysis would also require energy and power budget for the sensors, in particular for wireless sensors, which require ad hoc power supply. For actual space projects, the question is even more critical, requiring specialized analyses during all the global design loop.

6.3 Model development and Digital Twin structure

After preliminary analysis and sensor selection, the actual development can begin.
This section covers both the model, built on MSC Adams, both the middleware, coded on MATLAB/Simulink. A description is provided. Although the software development process is not the focus of the work, some insights are still provided.

It must be noted that the process may vary dependent on many factors defined during the preliminary analysis. Both the architecture, both the model and software choices can cause divergences in the method, which is not unique.

6.3.1 Adams model description

Creation from CAD assembly

Firstly, the model for simulations is briefly described. It has been built on MSC Adams starting from the CAD model of the prototype, created by Binetti.[1] The CAD model is an assembly composed by parts files, and included stock components, such as screws and washers. It has been exported from Solidworks, using '*Motion Study*' tool and the possibility to export into ad Adams file (.bin file). All the motion variables introduced on Solidworks were however been removed in Adams, in particular motions and constraints.

Gravity has been set as Earth's one, since the prototype is tested in laboratory and not on the Moon. Specifically, the value is -9806.65 mm/s^2.

Materials

Five materials have been defined and here listed, with application details:

- *Steel* has been used for stock elements such as screws, washers and nuts. It is the Adams standard for steel and is isotropic.
  - $\rho = 7.801\, g/cm^3$ density

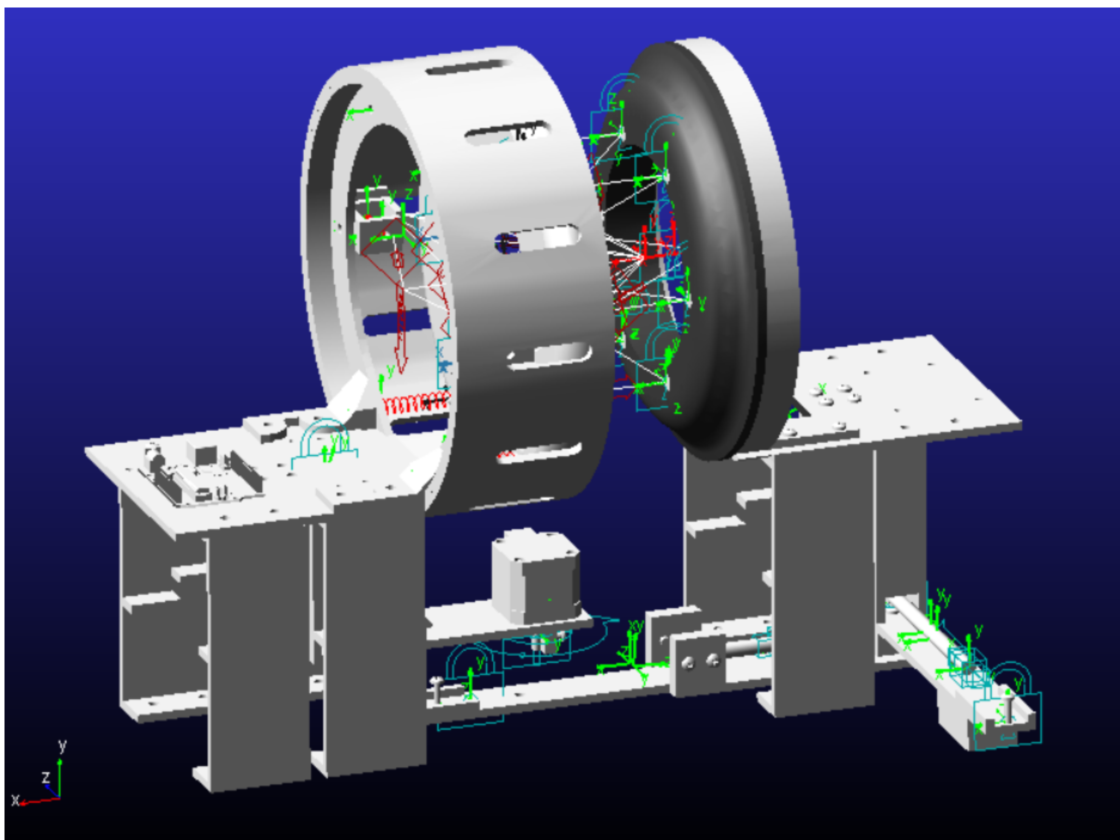

Figure 6.6: Image of the complete Adams model. Notice the reference triad in the low left corner. At left there is the Target, with the support structure; at right, the Chaser (black docking interface is visible).

  - $E = 207\,GPa$ Young modulus
  - $\nu = 0.29$ Poisson's coefficient

- *TPU (Thermoplastic polyurethane)*, a black thermoplastic polymer, halfway between hard plastic and rubber. It has been used for the docking interfaces, both male (chaser, visible in the picture), both female (target). Properties has been taken from Binetti's thesis: see [1] for more properties.
  - $\rho = 1.2\,g/cm^3$ density
  - $E = 0.0012\,GPa$ Young modulus
  - $\nu = 0.45$ Poisson's coefficient
- *PLA (Polylactic acid)* for all the other 3D-printed parts of the prototype, that is the majority of it.
  Density is not uniform already in the filament form (before printing), with differences due to brand and specifications; moreover, there is much more variability due to infill density and printing pattern. Therefore, a unique value for PLA does not exist,

nor an average value had been specified by Binetti. So, there exist two possible ways. The first would be measuring volume and weight of components, and compute their mean density; a generalisation could be done by computing the density for one part and using its value for all PLA-made components. Not needing so much precision at this level of design, it was chosen a second way: values were taken by sources and averaged to have approximate parameters.

  - $\rho = 1.24\, g/cm^3$ density [64]
  - $E = 3.475\, GPa$ Young modulus (at room temperature) [65]
  - $\nu = 0.375$ Poisson's coefficient.[66] Even though the material is non-isotropic for 3D-printing structure, for simplicity it was defined isotropic in the model.

- *Aluminium* has been used for specific parts: Arduino, batteries and the stepper motor. It clearly is a simplification, since these components are made of various materials and not filled internally. However, aluminium is halfway between heavy metals and lighter plastic materials and was used. Note that all these parts do not move and so don't contribute to dynamic actions. The Adams standard Aluminium was employed.
  - $\rho = 2.74\, g/cm^3$ density
  - $E = 71\, GPa$ Young modulus
  - $\nu = 0.33$ Poisson's coefficient.

- *Neodymium* was used for magnets, six on the chaser and six on the target. *N45* magnets were employed, so values were extracted from sources.
  - $\rho = 7.475\, g/cm^3$ density [67]
  - $E = 160\, GPa$ Young modulus [67]
  - $\nu = 0.24$ Poisson's coefficient.[68]

- The two *linear guides* have been treated with a different approach, being important moving parts. Inertial features (mass, density, inertia tensor and axes) have been extracted from Solidworks, since these components were imported from a pre-existing database.

Elements and groups of elements

Since each component must be correctly constrained, for a streamlined and lighter model many components were merged into solid groupings, appearing as a unique part. The prerequisite is to be tightly jointed components. For example, screw-joined components can be easily merged as one.
Not to lose inertial information derived by the materials assigned to each part, the *Aggregate mass* function was used to define the merged parts with the correct inertial properties (mass, axes and tensor of inertia).
Some minor elements such as screws and washers are hidden in the prototype visualization to lighten the model, but their properties are still included. All the resulting definitive parts are listed and briefly described in table 6.2.

Part	Nr. of units	Description
ASM_Chaser_1	1	The structure of the chaser, which moves towards the target.
DOCK_ALN_Male	1	TPU male docking interface, mounted on ASM_Chaser_1; it matches with the female interface. It is a single component.
FEMALE_DOCK_ALN_Female	1	Female TPU docking interface, which adhere to the male interface. It is linked by springs to TARGET_structure. It is a single component.
ground	1	Adam's default part representing the fixed reference for the model.
HAB_Guide_Mount	1	The support for Linear Guide 1 (the one for translational motion).
M_Nema_17_1	1	The stepper motor giving the misalignment to HAB_Guide_Mount and to all the chaser. It is mounted on TARGET_structure (constraint forces). It is a single component (already in the CAD model).
N45_Magnet_Chaser	6	Neodymium magnet inserted in DOCK_ALN_Male. It is a single component.
N45_Magnet_Target	6	Neodymium magnet inserted in FEMALE_DOCK_ALN_Female. It is a single component.
PR_Interface_2	1	Attached to the moving part of Linear Guide 1 (therefore it translates during the approach phase), it is the ground support for Linear Guide 2.
S20_100_38_B	2	Fixed parts of the Linear Guides. It is a single component (already in the CAD model).
S_20_LIFT	2	Moving arms of the Linear Guides. It is a single component (already in the CAD model).
TARGET_structure	1	Fixed structure of the target. It is the fixed section of the prototype in the absolute reference frame (ground). It also includes the electric and electronic circuit, as well as Arduino Uno module.
VERTICAL_Support	2	The two halves of the circular support for the FEMALE_DOCK_ALN_Female. It is linked by springs to TARGET_structure.

Table 6.2: List of all model parts (created by aggregating actual CAD parts). All parts group more than one component, except when mentioned.

Springs

The prototype includes six horizontal springs: four springs, along the principal direction of motion, link the female interface to the structure of the target; other two, horizontal but perpendicular (consider them along z direction in figure 6.6), join the two lateral supports to the target's interface. All the six springs hav been designed to offer support for the female interface but consenting partial mobility.
As from Binetti's indications, spring properties are:

- $D_{sp} = 13\,mm$ is the mean diameter of the spring;

Figure 6.7: Rear detail of the female docking interface. Notice the position of springs (two longitudinal and one perpendicular are visible) and the absence of screws, nuts and washers in the visualization.

- $d_{sp} = 1\,mm$ is the diameter of the wire;
- $n_{sp} = 10$ is the number of active coils;
- $k_{sp} = 0.5\,N/mm$ is the spring's stiffness.

Each spring has a mass of approximately 2 grams; the mass of the target's moving parts is about 0.975 kg. Their ratio is $\frac{m_{spring}}{m_{target,moving}} \sim 0.2\%$, allowing to neglect springs masses, which springs can therefore modelled only as forces, ignoring them as rigid or flexible bodies.

Some data must be included in the model to dynamically describe the springs. Here it is the analysis for the four longitudinal springs.
The four springs are in parallel, therefore stiffness and damping can be summed for an equivalent spring:

$$k_{eq} = \sum_i k_i \quad , \quad c_{eq} = \sum_i c_i$$

The springs are however defined as four different linear forces in the model, not as one single. Therefore each spring simply maintains its individual stiffness, with the mentioned value of 0.5 N/mm.
A different approach is for damping, which derives from other properties and can be computed as a fraction of the equivalent 'big' spring's damping. This value, having four equal springs, it is obtained as $c_{eq} = 4c_i$; reversed, it gives $c_i = \frac{c_{eq}}{4}$.
Moreover, the formula for the corresponding spring's damping (in absence of a real damper) is $c = \zeta c_{critical}$, with $c_{critical} = 2\sqrt{mk}$; m and k are mass and damping of the spring.
Merging all the formulas, one obtains $c_i = \frac{1}{4}\zeta\, 2\sqrt{m_{eq}k_{eq}}$ as estimated damping for a single

spring.
Considering: $\zeta = \frac{c}{c_{crit}} = 0.005$ for steel springs, as from [69]; $m_{eq} = 0.621kg$ as the mass linked to the four springs (female interface with its four magnets); $k_{eq} = 4 * 0.5\, N/mm = 2000\, N/m$. It is then obtained:

$$c_{i,long.spr.} = \frac{1}{4} 0.005 \cdot 2\sqrt{0.623 \cdot 2000} = 0.0881\, Ns/m = 8.81\, 10^{-5}\, Ns/mm$$

The other two springs, conversely, have been substituted during the reassembly of the prototype at the beginning of this project. The two new springs have the values of:

- $D_{sp} = 13\, mm$ is the mean diameter of the spring;
- $d_{sp} = 1\, mm$ is the diameter of the wire;
- $n_{sp} = 3$ is the number of active coils;
- $k_{sp} = 1\, N/mm = 1000\, N/m$ is the spring's stiffness; this value has been estimated by computing the elongation, by placing a 0.2 kg weight on it.

Moreover, each spring 'sees' one half of the lateral support. The two springs are not acting on the some body, and thus they are not in parallel. The value of damping is therefore estimated as:

$$c_{sp,tangential} = \zeta\, 2\sqrt{m_{tot} k} = 0.005 \cdot 2\sqrt{0.22 \cdot 1000} = 0.15\, Ns/m = 1.5\, 10^{-4}\, Ns/mm$$

Since the two lateral springs in the static initial position of target's vertical support (the two halves of the halo enclosing and sustaining the female interface) are slightly under traction, a *preload* has been added. Spring force is computed, by Adams, as

$$F_s = -c\frac{dr}{dt} - k(r - LENGTH) + PRELOAD$$

with r the distance between the two extremity markers, $LENGTH$ the reference length at default position and $PRELOAD$ the preload resulting from this position not being the spring's equilibrium state (characterized by the natural length L_0). With this formulation and setting $LENGTH$ as *'Default length'* (Adams takes the distance the distance in the geometrical configuration as the initial state), the $PRELOAD$ gets the expression:

$$PRELOAD = -k\,(LENGTH - L_0) = -1\, N/mm\ (23 - 17)\, mm = -6\, N$$

The length values were directly measured on the mounted prototype.

The static rest state of the prototype is not the theoretical one, since the vertical supports and the female interface slightly fall towards the ground, specially one of the two halves. Indeed, their neutral position depends on friction, mutual contacts and springs stiffness and depends on asymmetries of the system: the parts slightly lean toward the ground.
To reach a compatible also in the model, an *initialization* is necessary. Static equilibrium cannot be reached (MSC Adams reports an error), since the solver is not able to reduce to

zero the oscillations deriving from the springs in an acceptable time. A different strategy is therefore to use *dynamic relaxation*: running a first dynamic simulation without external motions from motors, letting the oscillations to dampen (even if not zeroing). At the end of the simulation, the system is not still but has minimal oscillations: this final state is frozen and conserved as starting point for every successive simulation.

Magnets

Six magnets on the chaser and six on the target are used for hard docking, that is for blocking chaser and target together. Made of *neodymium*, as said in section 6.3.1, the properties are:

- $F_{m,ref} = 105.7400\,N$ the force between two magnet when in close contact has been taken has reference;
- $D_m = 10\,mm$ is the diameter of each magnet;
- H_m is the thickness of each magnet.

More about the magnets is found in the work by Binetti.[1]

For simplicity, only six forces were modelled, to represent the main interactions between each couple of facing magnets. Otherwise, to also include all the secondary forces (between one magnet on the chaser and one on the target but not facing), it would have been necessary to model a total of $6 \cdot 6 = 36$ forces, giving the model more heaviness than accuracy. Moreover, the forces between magnets of the target are not considered, since they are mounted in the same rigid body and do not have any effect, not even minimal; the same for the chaser.
Another approximation is that the formula used for the forces considers axially-aligned magnets. As known, target and chaser are often misaligned (γ and Δ), and so the magnets are. However, it is the magnetic force itself to move the target interface towards the chaser when misalignments are present. Therefore, it is considered acceptable to introduce such an approximation, which was already adopted during the prototype's design.
All forces, bidirectional, are modelled with:

$$\begin{cases} F_m = -\frac{3}{4} F_{m,ref}\, D_m^2\, h_m^2\, \frac{1}{x_h^4}\,, & x > \tilde{x}_h \\ F_m = F_{m,ref}\,, & x \leq \tilde{x}_h \end{cases}$$

The first equation is far-field, acting when the magnets are more than $\tilde{x}_h$ away, with $\tilde{x}_h = \sqrt[4]{\frac{3}{4} D_m^2 h_m^2} = 5.097mm$. Distance x_h is measured between the centre of magnets' faces, reucing to zero at contact.
By way of example, the force between one of the six couples of magnets has the following syntax in Adams:
$-IF((.Prototype_from_Motion_Study.Xh_Magnets5 - (3/4*0.01**2*0.003**2)**(1/4)) : 105.4700, 105.4700, (3/4*105.4700*0.01**2*0.003**2*(.Prototype_from_Motion_Study.Xh_Magnets5)**(-4)))$

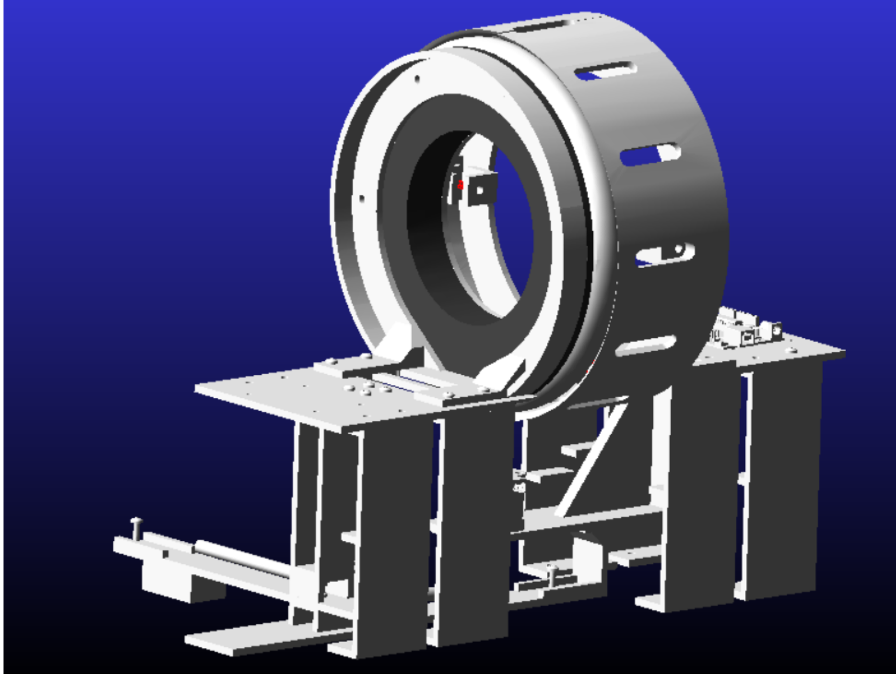

Figure 6.8: Docked view of Adams digital model.

Contacts

For specific classical constraints (revolute joints, fixed parts, translational joints,...), it was possible to define kinematic constraints between parts. However, other constraints need specific forces when degrees of freedom are not completely suppressed.
Therefore emerges an important class of forces, the *Contact*, which has multiple purposes:

- Modelling impacts, especially for the docking event;
- Avoiding permeation between parts;
- Allowing for parts to lean against other parts, in substitution to other specific constraints;
- Modelling friction in all its forms.

Contacts has been modelled as *IMPACT forces*, which has the following equation:

$$F = \begin{cases} 0, & \text{if } q > q_0, \\ k(q_0 - q)^e - c_{\max}\dot{q} * \text{STEP}(q, q_0 - d, 1, q_0, 0), & \text{if } q \leq q_0 \end{cases} \tag{6.1}$$

q is the distance between bodies, while q_0 is the trigger distance. The theory behind Adams *IMPACT* is:

- It is a non-linear Hertzian model. Hertzian contact theory is built on elasticity theory and based on the curvature radius of the bodies.

- Among all the possibilities to model contacts, it is a fast and stable option, even though it anyway introduces slowness in the simulation.

- Parameters are:

  – Stiffness of the contact k, which approximates both the geometrical interactions between the bodies, both the material properties;

  – Damping c_{max}, which acts as numerical damping and is not typical of Hertzian theory, nor of the physical contact; it should be highest possible to reduce simulation time but without affecting the physical behaviour;

  – Hertzian exponent e, which determines the shape of the force, is generally comprised between 0.5 and 2.2.

  – Penetration depth d_{max}, the maximum permitted value of numerical penetration.

  – *Friction*, which can be modelled in different ways and with some parameters such as *transition velocity.* Alternatively, settings on the elasticity of the collision

The issue is to determine such parameters, which generally do not have an univocal value. An help came from [70], as a theoretical basis. As an example, values for the Hertzian exponent are: $e = 1.5$ for soft metals, $e = 1.4$ for PLA, $e = 1.2$ for TPU. However, values are often at the discretion of the modeller and not a universal truth. Contacts were thus defined depending on the materials involved, as:

- TPU-TPU: e=1.2, k=10^8 N/m, $c = 0.001k = 10^5 Ns/m$, d_{max}=0.1; it is the contact between the two docking interfaces.

- Neodymium-Neodymium: e=1.5, Neodymium is a soft metal, used for the magnets (which are not completely made of Neodymium, but it is an approximation).

- TPU-Neodymium: e=$\frac{1.5+1.2}{2} = 1.35$, k=10^8 N/m, $c = 0.001k = 10^5 Ns/m$, d_{max}=0.01mm

- PLA-PLA: e=1.4, k=10^8 N/m, $c = 0.001k = 10^5 Ns/m$, d_{max}=0.01mm

- TPU-PLA: e=$\frac{1.2+1.4}{2} = 1.3$, k=10^8 N/m, $c = 0.001k = 10^5 Ns/m$, d_{max}=0.1mm

- For contacts, Adams' standard Coulomb friction was added.

System of reference

The reference fixed triad is:

- *x axis*, along the motion direction, pointing from the Chaser to the Target (it is fixed and does not move with misalignments).

- *y axis*, vertical.

- *z axis*, transversal, result of the right-handed triplet.

For the misalignments:

- Δ is positive along z axis.
- γ is positive around y axis (right-hand rule).

Both can be visualized positive towards the right, when facing the target, with back to the chaser. See figure 6.6 for a visualization of the triad.

Actuators constraints

The motion has been modelled with the Linear Guide's limits, in order to prevent exiting from the range. It has indeed 100 mm of motion.

Limits to γ have been computed, since lateral screws obstruct the rotation of the HAB_Guide_Mount. From geometric relationships, the angle's range is computed to be $\pm 11.325^o$, while inside the model it was inserted $\pm 11.32^o$.

For Δ, the range is $\pm 20\,mm$.

In search of real-time: lightening the model

The main obstacle is that this complete model is too slow for real-time applications, at least with the available hardware (a student's personal laptop). A lighter version is necessarily needed. Besides, a model requiring lighter hardware is an asset also for space applications, in which hardware has weight and power limitations.
The first attempt of lightening the model has been:

- Deleting the less relevant contacts, in particular those involving parts who are unlikely to interact;
- Removing unused variables;
- Removing friction between orthogonal impacts.

A second, more drastic, attempt has seen an additional removal of contacts.
However, even these simplified (and less faithful) models resulted inappropriate for real-time applications. Even changing solver parameters in search of better computational performance, the best result has been $\alpha = \frac{t_{simulated}}{t_{wall-clock\,(CPU)}} = \frac{3\,s}{14\,s} = 0.214$; it means requiring 14 seconds to simulate 3 seconds of prototype's behaviour. Clearly, this result is insufficient for achieving real-time ($\alpha > 1$, as seen in section 7.3.1). To be transparent, the best result was $\alpha = \frac{t_{simulated}}{t_{wall-clock\,(CPU)}} = \frac{3}{4} = 0.75$; in addition to be still not satisfactory, this condition was achieved with solver settings that gave terrible results in modelling the behaviour at impact: this direction is not feasible.
Furthermore, Simulink-Adams communication was not yet considered in this computation, perhaps resulting in an even slower time flow. A different strategy proved clearly necessary.

6.3.2 Real-time solution: the surrogate model

A new solution was offered by *surrogate models*, already outlined in section 5.1.4. Since results readiness is basically preferred to accuracy (within certain limits) in real-time applications, *surrogate models* represent a better approach, compared to the slow multi-body simulations.

The switching from the complete multi-body model to a simple surrogate polynomial model has primarily the function to scale down numerical complexity, for a fast algorithm, often consisting of few matrix operations. A secondary benefit is that now the model can simply be implemented on MATLAB/Simulink, also eliminating the need for Simulink-Adams co-simulation.
Indeed, the architecture initially considered to link MSC Adams to Simulink through *Adams Control Plant*, an Adams tool specifically designed to run co-simulations with Simulink. An alternative would have been *FMI* format; details about *FMI* are in section 7.3.1.
The reduction to a simple formula consents to reduce the complexity of the middleware, with less development efforts. Moreover, bypassing the Adams-Simulink communication further streamlines the middleware, speeding up its cycle, theoretically allowing for shorter control cycles.
Summarizing, the reduction in simulation time mainly arises from the drastic simplification of the model, but also the missed Adams-Simulink communication has a role in enhancing the time performance of the simulator, at the expense of some accuracy.

Design of Experiments (DOE)

Design of Experiments (DOE) methods involve models with multiple inputs, which affect a spectrum of resulting outputs. These methods investigate how variations in the inputs influence the output of the model.
The most common applications are:

- *Experimental campaigns* planning of *experimental campaigns* is the one application, as the acronym itself suggests. Evaluating a real system rather than its model, considering the different inputs to estimate their impacts on the physical phenomena, is central in the definition of rigorous tests. Also, exploiting equivalent models for preliminary simulations can help this process.

- During the design phase of any project, it is crucial to conduct a campaign to evaluate how certain quantities evolve in response to varying others. DOE an be an instrument for identifying the best candidates for specific hypotheses. Here, we enter the field of *optimization*, for which DOEs method can build a batch of simulations to be run exploring the input variables range. However, prototypes are often a more accurate solution to evaluate input-response correlation heuristically.

- *Surrogate models* are another field in which DOE is employed. Again, after defining the ranges in which each input variable could vary, these are combined to create a batch of m sets of initial conditions for the original model.
  As later explained, the goal is to cover the range of inputs as comprehensively as

possible. Indeed, the greater the available data set, the more valuable the model's approximation. Please be aware that there exist cases is not possible to obtain a surrogate model with adequate accuracy, due to inherent characteristics of the involved phenomena.

DOE methods start by defining valid ranges for input variables, from which an assortment of initial conditions configuration is obtained. In other investigation methods, one variable at time is varied over its span, with the others are kept fixed, in order to evaluate the influence of the changing one on the system or the model. Instead, DOE methods consider to variate all the inputs simultaneously; the goal is to study the dependence on all the variables together, both to accelerate the study, both to evaluate the coupling effects of varying inputs.

Consider n input variables: $u_1, u_2, ..., u_n$. During one single test (experimental or simulated), the initial configuration is $u_{11}, u_{12}, ..., u_{1n}$. Similarly, it is possible to continue to a define a second initial vector of conditions for a second try, then a third, and so on. Therefore, by considering m different tests, each with a specific set of initial conditions, the *design matrix* takes shape:

$$\begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ u_{21} & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m1} & u_{m2} & \cdots & u_{mn} \end{pmatrix}$$

The purpose of DOE is to construct an effective matrix, which must explore all the ranges of variables, perfecting their coupling, maximizing the effectiveness of the m available tests. Clearly, the higher m is, the most accurate the study is.

Response Surface Method (RSM)

If an input variable is linked to an output variable, their a correlation, under formal conditions, is called *function*; if the inputs and the outputs are more than one, one can talk about *surfaces. Response Surface Methods (RSM)* analyse these connections between the set of inputs $u_{1:n}$ and the set of outputs $y_{1:l}$.
The combination of this investigation with DOE methods exploits the best of the two, for optimal results. One of the applications is identifying a polynomial approximation of the model, linking the outputs, or *responses*, to the set of inputs. The fitting may be done with different procedures, such as least square method, and allows to mould a simple approximation of a numerical large model. Devising a good DOE enhances the quality of the resulting fitting, with the same number tests performed; thus, it is one way to create a *surrogate model*, as seen in section 5.1.4.

For this work it has been estimated a *quadratic polynomial* for each response. It means that any response y_k is approximated by the formula (c are the coefficients):

$$y_k = c_0 + c_1 u_1 + c_2 u_2 + ... + c_n u_n + c_{1,2} u_1 u_2 + c_{1,3} u_1 u_3 + ... + c_{n-1,n} u_{n-1} u_n + c_{1,1} u_1^2 + ... + c_{n,n} u_n^2$$

The generalized formula is:

$$y_k = c_0 + \sum_{j=1}^{n} c_i u_i + \sum_{j=1}^{n} \sum_{h=j+1}^{n} c_{ij} u_i u_j + \sum_{j=1}^{n} c_{ii} u_i^2$$

The polynomial should also show an additional term, ε, representing the error between the approximation and the model (or the experimental data). ε needs to be minimized, but generally it is not possible to define it, except for the test points used for building the model. This means, as further ahead discussed, that ε is available for accuracy considerations but not for validation of the model, which shall be performed on a set of tests not used for building the surrogate model.

Now consider the current case. For the DOE sampling used to build the design matrix, by selecting the m different sets of initial conditions, it was employed the *latin hypercube* method, often at the basis for Montecarlo analyses. *Latin Hypercube Sampling (LHS)* statistically generates a simil-random distribution, converted in meaningful values for the n-inputs initial configurations of each of the m simulations. This way, it is possible to build the $m \times n$ matrix, the two-dimensional space called *design matrix*.

In random sampling, each sample is taken independently from the others and not from any specific subset of the whole set.[71] This may lead, especially for small numbers of simulations m, to a net of samples with variable mesh sizes, denser in some areas and sparser in others. One problem is that some small subsets of values (peculiar sets of initial conditions) with high consequences on the simulation or on the system's behaviour, may be inadequately covered by the sampling; this is more true at the reduction of the number of samples m. Therefore, a sampling techniques that can maintain the randomness of the sampling, but with a certain uniformity of the mesh, is preferred.

Latin hypercube sampling also is a probabilistic procedure, but starts by subdividing the whole set in exactly m sectors, as the number of tests. Then, the random operation starts: for each input $u_{1:n}$, m values are defined, each one randomly picked inside on of the m sectors. Supposing a $0 < u_1 < 1$, for example, a first pick for u_1 is randomly picked between 0 and 0.1, then one randomly picked between 0.1 and 0.2, and so on, and the same for $u_{2,\ldots,n}$.
The third step is to randomly sort this sample vector for u_1. The vector $u_{1,1:m}$ contains a simil-random distribution of samples, that however uniformly cover all the range in which u_1 can variate. The same is done for all the other inputs $u_{2:n}$.
The final action is to combine these vectors, in order to create the $m \times n$ design matrix. This allows to have m rows, each one containing an apparently randomic set of inputs. Although, globally, the input set is investigated in the most uniform way possible. *Latin hypercube* algorithm, qualitatively described, allows to exploit sampling randomness, but forcing it to uniformly cover the whole set, without involuntary holes.
Figure 6.9 shows a visual example of LHS, from [71]. See how each vertical sector and each horizontal sector contains a sample. Here, it is the case of two inputs (u_1 and u_2), one on the x-coordinates and the other on the y-coordinates: n is 2. Five samples are taken, therefore $m = 5$. This would be translated into a 5×2 design matrix.
Please refer to [72] for a formal definition of LHS, instead.

Adams Insight actually uses equally spaced points, reducing the randomness factor. However, as explained, the n inputs and m sets of initial conditions are randomly combined to create the sets of initial conditions of each test: each of the m combinations takes one random value among the m equally spaced samples for u_1, one random value among the m equally spaced samples for u_2, and so on up to u_n. None of the values in which an input has been varied is unused nor used two times: the sample sets of each u_i are independently permuted, with no repetition.

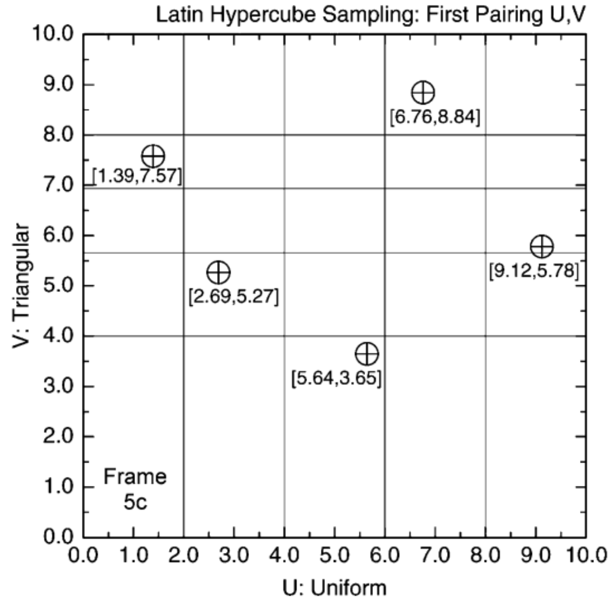

Figure 6.9: Visual representation of an exemplifying Latin Hypercube sampling over a 2D space, with *m=5*. On the x-coordinates a uniform segmentation has been performed, while on the y-axis the sectors show different weights.[71]

Inputs and responses

For the DOE, and thus for the simulation campaign, the following input vectors have been selected, giving $n = 4$:

- *Angular misalignment* γ, with range [-11.32, 11.32] degrees;
- *Linear misalignment* Δ, with range [-2, 2] cm;
- *Initial relative position*, which is the initial distance between chaser and target, with range [0, 0.1] m;
- *Chaser speed*, in the range [-4, 4] cm/s, with positive values for approach, and negative values for backward retraction; velocity is approximated constant during the simulation, but zeroes if arriving at limit switch (actually it reverses, in reality, but the control algorithms should stop it when identifying contact).

An important to being able to iterate the surrogate model, by using the outputs as new inputs for a new cycle. This allows the user to use the surrogate, which predicts the responses after t_{surr}, to have predictions after longer time, as long as it is a multiple of t_{surr}. γ and Δ are considered constant and thus not taken back from outputs to inputs, while the other two outputs can be taken as inputs for a second iteration of the surrogate. Responses (simulation outputs) have thus been chosen as:

- *Relative distance* between target and chaser, at the end of the simulation, in magnitude value;

- *Chaser speed*, in magnitude value, at the end of the simulation; it should be equal to the initial value, unless zeroing if arriving at limit switch;

- *Female acceleration*, root mean square of the magnitude value over the simulation; it gives guidance on the force exchanged during the impact. Differently from the other two properties, it is not useful for the control strategy, being just a first approximation of the contact dynamics.

This means having 3 response polynomials.

Adams Insight

While a custom-made surrogate model is always possible to build, Adams offers a useful tool that facilitates the creation of surrogate models of their complete multi-body equivalent. First of all, this tool has been used on the complete model previously described, not on one of the lightened versions, which are faster but less accurate.
Speaking about Adams Insight, it is a GUI platform that allows to plan a DOE (Design of Experiments), creating the design matrix based on user's indications; then, it automatically launches the resulting campaign of multi-body simulations on Adams and fits the results to create the polynomials. Moreover, the possibility to export them into MATLAB format has been exploited to facilitate the integration of the surrogate model inside the Digital Twin architecture.

If this is the procedure used, Adams Insight also offers other methods and other possibilities, such as optimizations and parametric studies.

Simulations specifications

Each one of the m simulations runs the model for 0.05 seconds (simulated time); short simulations are considered, since the surrogate will operate on short time-spans, on a 0.1-1 second scale (as of a preliminary estimation). The surrogate may then simulate for multiples of 0.05 seconds, by iterating the formula, feeding the outputs back into the inputs.
For the campaign, $m=400$ simulations were performed. More technical details: GSTIFF solver was used, with C++ solver and multi-thread mode.

Accuracy considerations

R^2, the coefficient of determination, is a number between 0 and 1, measuring data variability (how much values diverge from the average value).

$$R^2 = 1 - \frac{\sum_i (\tilde{y}_i - \hat{y}_i)^2}{\sum_i (\tilde{y}_i - \bar{y})^2}$$

y is the considered variable. In particular: $\tilde{y}_i$ is any measured (or simulated) point; $\hat{y}_i$ is the relative estimated value; $\bar{y}$ is the mean value. It is a good instrument to measure how well the surrogate model fits the set of data; note that it is applied to the data used to build the model (the outputs of the 400 simulations), not on a new set of points, taken as validating tool.
For each response, the simulations give a set of points (inputs and the response), used to build the fitting polynomial. Each of the three responses is thus considered as an output of the complete Adams model ($\tilde{y}_i$), but also the mean of such values ($\bar{y}$) is computed, as well as the estimated outputs provided by the approximating polynomial (given the same starting conditions $\hat{y}_i$).

For the surrogate model, the results, directly computed by Adams Insight, initially were:

- $R^2_{relative\,distance,\,final} = 0.365$: it is a low value, for the relative distance between chaser and target at the end of the simulation.
- $R^2_{chaser\,speed,\,end} = 0.898$: this value was quite high right away.
- $R^2_{target\,acceleration,\,RMS} = 0.238$: low value for the dynamic estimation of the contact.

However, an analysis revealed how few values spoiled the model. This consideration helped to enhance the model accuracy, by performing a first *outlier cleansing*. The process is described in the next section; here, the resultant surrogate gave the R^2 values here listed:

- $R^2_{relative\,distance,\,final} = 0.831$: a drastic increase means a very effective data cleansing. A clearly bad approximation turned into a possible good one; an evaluation of its validity is in section 7.4.
- $R^2_{chaser\,speed,\,end} = 0.931$: also this second surrogate sees an improvement, passing from a good R^2 value to a very good one; again, this is still not a sign of a certainly reliable model.
- $R^2_{target\,acceleration,\,RMS} = 0.350$: despite an improvement, this value is still low.

One consideration can be done: the physical model is strongly linear until the impact comes into play. During the collision, phenomena are not linear, and many quantities drastically change. So, the distance is a quite linear property and can be easily modelled by a polynomial, however considering some struggles when it turns zero in a quasi-discontinuous manner at collision. Same story is for the speed.

Differently, acceleration (which actually is a force) is not simple to approximate. It involves friction, discontinuous phenomena, angles of collision, springs; moreover, all of this only happens under specific initial conditions, which brings to the contact. The result is a set of points often null and non-null only when docking happens, with a considerable dispersion, not indicated for a simple polynomial fitting.
As a result, the surrogate for acceleration was discarded from the beginning, not to add worthless computations to the Simulink real-time cycle. For better dynamic estimations, it will be possible to refine the original Adams model, and perhaps directly launch it at the end of the run, to have a first approximation of the forces at play.

It is again important to note that R^2 values are built on the set of simulations used to build the model; therefore, even if R^2 is high, it cannot be considered as a validation, which requires other cases as proofs. Validation has indeed its own section, 7.4, and is not good as desired.

Data cleansing

The surrogate models were refined using Adams Insight, which allows various inspections over the results of DOE and simulation campaign. This enabled to individuate some runs that gave anomalous results, spoiling the fits. These *outliers* are at the extremes of response data distributions and often even outside the acceptable range. For the following consideration, *studentized residuals* (residual normalized by the estimated standard deviation) were used, as computed by Adams Insight.

First of all, p_{end} saw a great benefit from the removal of outliers, as mentioned. A first try saw the removal of all simulations that took with a studentized residual greater than ± 3.5; two simulations were identified (*127* and *152*, considering $1 \leq m \leq 400$) and removed, bringing to $R^2 = 0.676$. Starting from the new obtained fitting, the same procedure was repeated, and then another time. This removed respectively four and then six other runs from the interpolated set, with a final value of $R^2 = 0.896$. These outliers simulations were noted.
A second approach was to identify problematic runs. The allowed values for target-chaser distance p are $0 - 0.1\,m$. There were identified 22 runs with p_{end} exceeding this range, in the original set of simulations; the problematic runs reduces to 14 if considering a $\pm 5\%$ of error ($-0.005 - 0.105\,m$) and to 8 if considering a 10% (runs *34, 78, 127, 152, 246, 252, 340, 348*). These were noted.

For chaser's speed v_{end}, R^2 was already high. Two simulations over the 400 (*152* and *246*) had a studentized residual greater than ± 3.5.

Finally, two cleansing iterations were done for target's acceleration, each time removing the runs with studentized residual bigger than ± 3.5: eight simulations were identified. However, R^2 only rose to 0.468.

After identifying problematic runs for the three response surfaces, it was chosen to select the most relevant outliers among the pinpointed ones, to create the surrogate model to be used. For consistency, indeed, it was decided to remove the same outliers for all the three responses polynomials. The reason is that this creates a congruent 'reduced' fitting: a unique surrogate model for all the three responses. Using one different cleansing for each of the three properties would have brought higher R^2 values, but the three formulas

would not have been coherent one with the others.
The removed runs (of $1 \leq m \leq 400$) were eight over 400 (2%) were:

- *34*: this simulation affected both acceleration and distance;
- *78* affected only p_{end}, which fitting is important to refine;
- *127* affected both acceleration and distance;
- *152* was problematic for all the three responses;
- *246* also was problematic for all the three responses;
- *252* affected both acceleration and distance;
- *340* only affected p_{end};
- *348* affected both acceleration and distance;

A trend in these simulation was searched, to qualitatively understand the reason behind these problems:

- p_0: five of the eight outliers has a starting position of less than 2 cm. This may bring interactions between chaser and target difficult to be modelled by the multi-body model, such as interactions between magnets;
- γ is negative in all the outliers, and six of the eight cases have a value lower than -5°, which is outside the tested requirements of Binetti's work (section 4.1);
- Δ is again negative in all the outlier runs (both negative misalignments are towards the same 'side'); the values are however not particularly big;
- *Initial chaser speed* v_i is well distributed between positive and negative values, but many cases have quite high values (all but one outlier have speed higher, in modulus, than $2\,cm/s$). High speeds are not a problem during the approaching phase, with chaser and target far away, as these values are even used during the simulations; the problem may be if the chaser has high speed when close to the target.

The overall picture shows negative and high misalignments, together with considerable speeds. However, to well understand why these initial conditions led to problematic cases, the strategy would be trying to simulate them again with also the graphical interface, also evaluating if these simulations causes a jump in the time needed for the computations. Note that, of the 400 runs needed for the surrogate, few simulations required the majority of the time (probably the 95% of more of the time, but it is a rough estimate).
However, time limitations for the thesis project and the preliminary nature of it, did not allow to further investigate the reasons behind the outliers. This would be also the first step of a new iteration: refine the original Adams model, mostly in the definition of motion, and then create a new surrogate model. This is one of the most important refinements required by a possible first update of the Digital Twin.
For this preliminary version, the only other analysis performed about the model is the validation process, as in section 7.4.

6.3.3 Middleware structure

With the model ready, it is possible to move to a description of the middleware. In this chapter the middleware is considered, being the whole Digital Twin, except the digital model built with Adams but then integrated, as surrogate model, into the middleware. It has been moulded with Simulink.
Briefly, the process has been:

- To develop a precise block architecture, based on the preliminary analysis seen in section 5.2.
- To code, block by block, the middleware.

To facilitate the development, the middleware was divided in these logic blocks, the majority of which built as subsystems, in order to proceed gradually and debug step-by-step.

Overview

The middleware was entirely coded on Simulink, exploiting its possibilities of signal and time management: a paced simulation is used, to keep a 1:1 ratio between simulated time and real time. Many *MATLAB function blocks* were however added, taking advantage of MATLAB's broad capacities in data manipulation.
One weakness of Simulink is the possibility of using only a limited number of MATLAB functions in its *MATLAB function blocks*, since Simulink translates them into *C* code before compiling; therefore, sometimes emerges the necessity to recall external MATLAB functions in *extrinsic* mode.

Another critical point is Simulink-Arduino communication, which is possible in many modes, each with certain limitations. As seen later, the final strategy is: opening *serial communication* (which is not Arduino specific) with MATLAB, by exploiting the opportunity of running pure MATLAB code before Simulink compiles the code and starts counting time. This communication is used to upload the firmware on Arduino. Then, this communication is closed, the model is compiled and started, and so a new *serial communication* is opened, this time directly between Simulink and Arduino. More details comes later.

Passing to an overview of the middleware: it reflects the block scheme used for passing from functional and architectural analyses to a logical draft of the middleware, built to organize the coding work.
Figure 6.10 shows the final visualization of the middleware, as a model in Simulink GUI. Names are too small to be read, thus a description follows (more details about each subsystem are in the following sections):

- In the top-left corner there are simple instructions for a new user to start the model, while the other following instructions are shown as messages in the Diagnostic Window. Just below, there is the *configuration block*, which initializes Simulink-Arduino serial communication.

- At left, the brown block simply takes user inputs, prompted in the initialization phase, and inserts them as signals in the main model, with various measurement units. At its right there are display modules, for the suer to remember these values during the simulation.
- At right of the Display blocks, not highlighted with a colour, there is the *Serial Receive* block, which must be one unique for the whole model: it receives and reads messages incoming from Arduino (ASCII format).
- The purple subsystem sends some initial parameters to Arduino, to replace default ones.
- The yellow subsystem is for hardware initialization, ensuring correct homing of the chaser and initial misalignment imposition. Above it, two buttons (apologise for being a bit cropped) are used by the user for managing homing, which is done visually, as in the original project by Binetti.[1] A zoom of these two last subsystems is in figure 6.11.
- Finally, the green module is the main part of the middleware, which manages the real-time control cycle.

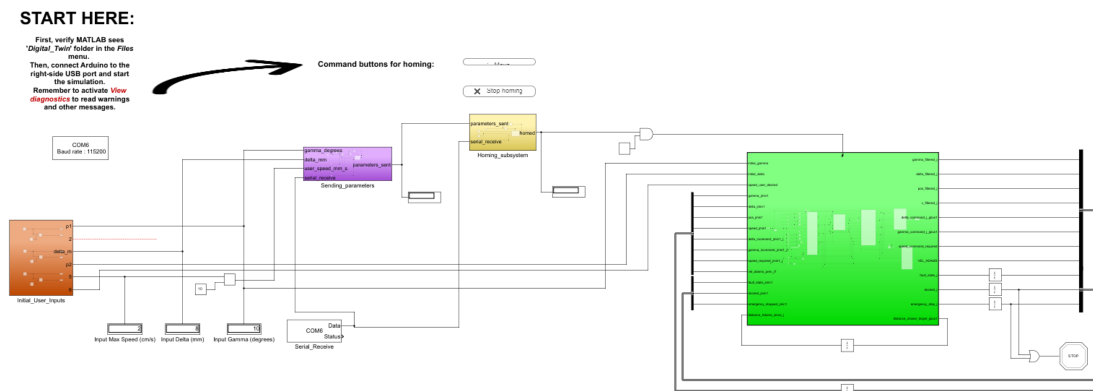

Figure 6.10: Middleware overview in Simulink. Colours have been used to facilitate subsystems recognition. Callback functions (MATLAB code launched before compiling) are not visible in the plot, being written separately, in another window.

Some details can be said about the real-time subsystem, as shown in figure 6.12. At top-left of the picture, there is the logic allowing to start the real-time phase, with a *pulse generator* activating periodically the main triggered subsystem. More details about each part are in the following sections; it is suggested to come back to this overview during the reading.
Also, the image clarifies the feedback logic, with outputs returning as inputs for the next cycle, with data buses used to simplify the visualization. Moreover, in the bottom-right side of the picture, there is the *Stop simulation* for ending the simulation under defined condition.

Now, a quite detailed and technical description of the modules follows.

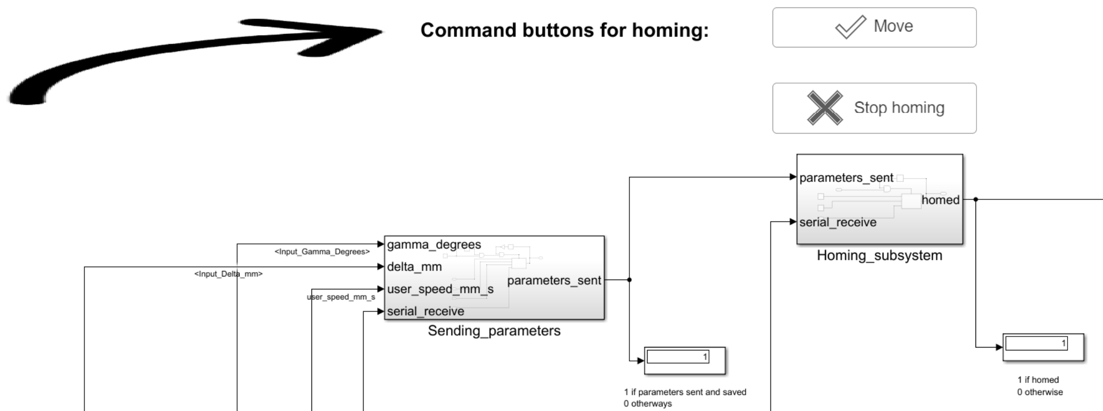

Figure 6.11: Detailed zoom of the middleware overview. The focus is on the first subsystems, activating after initialization with MATLAB functions and after model compiling, but preliminary to the switch to real-time.

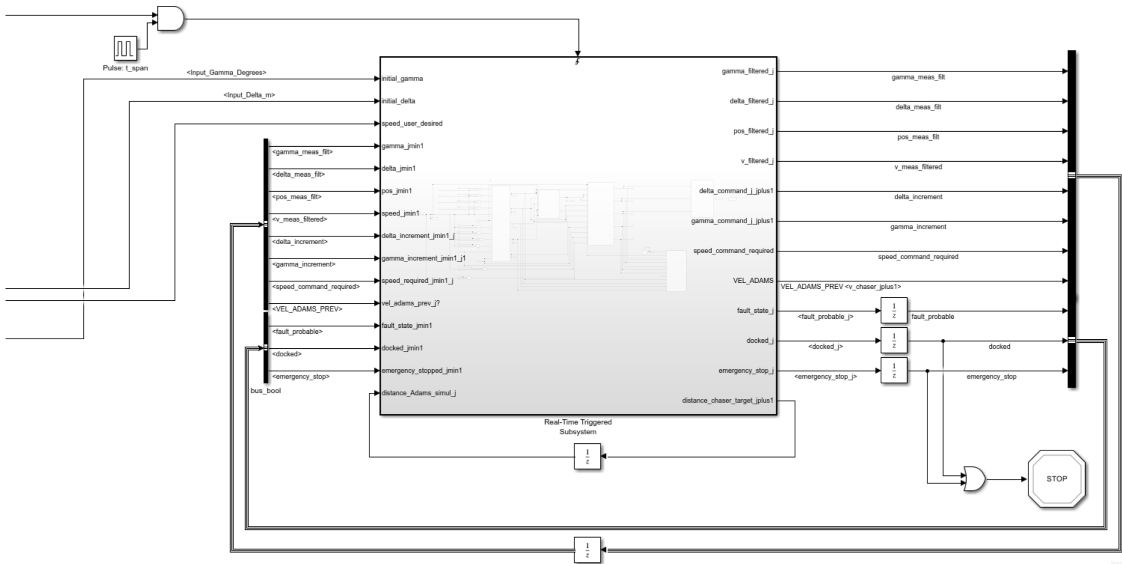

Figure 6.12: Detailed zoom of the real-time subsystem.

User inputs

Exploiting the possibility to run MATLAB code before the starting of the simulation, a MATLAB function asks the user to input:

- The initial angular misalignment angle γ, comprised between $\pm 11.325^o$, as seen in in the previous pages.
- The initial translational misalignment Δ, with value inside $\pm 20mm$.

- The desired approaching speed of the chaser, with a positive value not greater than $3\,cm/s$. This value is the the cruise speed tried to be kept during chaser's approach.

These queries are asked with the input prompt window in figure 6.13, which asks again in case of invalid values or characters. Moreover, the workspace is cleaned from pre-existing variables.
This parts of the code are inside the *Initfnc* callbacks window, which allows to run MATLAB functions just before model's compilation and before time starts flowing. There is also a block in the model, already mentioned, which takes these values, saved as MATLAB variables, and introduces them as signals inside the model, converting in SI units (radians for angles). See the brown module in figure 6.10.
Other parameters are then asked to the user in this initialization phase, later mentioned, are:

- The time-step to be used for the real-time loop;
- The firmware's version to be used and uploaded on Arduino.

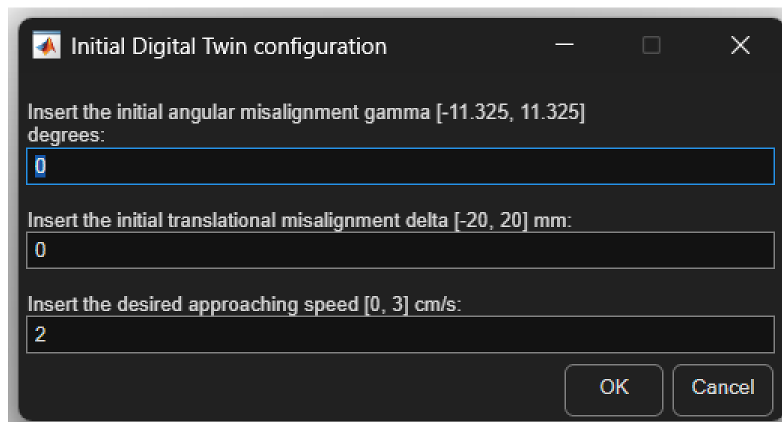

Figure 6.13: Window asking the user run settings.

Initial Arduino serial connection

When the user has prompted the values, the initializing MATLAB function continues, trying to connect with Arduino, connected by a USB cable to the computer. Useful warning messages are displayed in case of problems, such as reminding to connect the USB port, up to successful connection. This first connection is established through MATLAB's *serialport* function and is conserved only for the initialization of the model.

Uploading of the correct firmware

The initializing MATLAB function continues and:

- Calls *Arduino CLI (Command Line Interface)*, the executable file to govern Arduino, asking it to upload the correct firmware file on Arduino Uno board. This firmware

file is in a specific folder in the Digital Twin folder where also the middleware is: Arduino CLI needs to know the exact position to upload it.

- After having used Arduino CLI for the uploading, its communication is closed and *serialport* communication is opened.

- As already done, MATLAB sends a '*?*' query to check if the uploaded firmware is the correct one, by comparing Arduino's answer with the desired version, firstly specified by the user; the firmware contains its version in a stored string than can be only edited manually by the user on the file itself. In case of discordance, Arduino warns the user.

Re-initialization of serial communication

After the initialization phase, the communication between MATLAB and Arduino is closed, clearing the USB port. A new communication is established, this time between Arduino and Simulink. Indeed, for the rest of the simulation, it is important that Simulink directly reaches the firmware, without additional intermediate steps that could adulterate time management. For a simpler code, the whole Digital Twin uses the right-side port of the computer, when named *COM6*. It may happen that, due to various use of USB-port and connected devices, the PC does not recognize Arduino as COM6. It is recommended to change it into COM6 in computer's settings, rather than changing the *COM#* number in the Digital Twin.
Dialogue is made possible by the *Serial Configuration* block of Simulink's *Instrument Control Toolbox*. It was chosen not to use Simulink Support Package for Arduino, which is offered as a dedicated option. Indeed, it would generate C code directly from Simulink blocks, uploading it and replacing the predisposed firmware, making it difficult to manage.

Warning: the *Serial Configuration* block is critical, since it can become corrupted without clear reasons. If this happens, with the *Diagnostic Viewer* window reporting impossibility to connect to *COM6* port even when Arduino's port figures as COM6, one solution is to cancel and integrate again an identical *Serial Configuration* block. In this case, please insert the conditions:

- *Port*: COM6;

- *Baud rate*: 115200.

Sending initial conditions to the prototype

With the correct firmware on Arduino and the communication now held by Simulink, another step of the initialization is sending the initial inputs to the firmware. This part is not inside the MATLAB *Initfnc* block, which runs before ending the model compilation: if it was, during the passage from MATLAB-Arduino to Simulink-Arduino communication, these values would be reset to the default value, which generally is zero. Therefore, the first actual Simulink subsystem (purple in figure 6.10) is the one responsible of sending and verifying the reception of values, in addition to generically verifying Simulink-Arduino

connection.
The parameters sent to the firmware to replace the null default values are:

- Initial desired γ, angular misalignment, expressed in degrees;
- Initial desired Δ, translational misalignment, in mm;
- The tentative velocity of the Chaser with respect to the Target during the approaching phase, in mm/s; this value is, currently, not used by the firmware;
- The time-step value for the control cycles.

Since there is no possibility to impose these initial conditions in other ways than with the actuators, which are controlled by Arduino, it is necessary to insert them in the Digital Twin, which indeed is responsible also of the control of the prototype: Digital Twin and control software are integrated and are the same thing. Having them separated in two distinct and communicating unities is a more advanced and professional approach, not possible with the resources available for this work.

The *Enable* subsystem is activated when time starts flowing, at the end of MATLAB initialization function and compilation process; figure 6.14 shows the blocks inside of it. It receives the three parameters as inputs, by a separate subsystem which, as seen, introduces inside the model workspace the values prompted by the user during initialization; this separate subsystem, mentioned in section 6.3.3, also manages unity conversions. The blocks managing the connection are *Serial Send* and *Serial Receive* from Simulink *Instrument Control Toolbox*. They use the predisposed communication to send and read binary data. These blocks are outide the subsystem. Other blocks convert between numerical values, strings and binary values.
A MATLAB function analyses the incoming messages, sent by Arduino in response to the received and saved values, and closes the subsystem once it is certain that Arduino has received the parameters and sent back confirmation, displaying messages for the user. The output is simply a *0* signal turning to *1* at process success, activating the following subsystem.

Homing of the actuators

Before the actuation of the commands, it is necessary to bring the actuators to their neutral position. Since there is no sensor directly in/on the actuators, neither a limit switch, *homing* control is done with only the visual control of the user, as in the work originally conceived by Binetti.
The subsystem communicates with Arduino; firstly, it asks the user if motor 1 (linear motion) needs to be homed. If the answer is positive, it sends Arduino a request to move the actuator of 10 mm; then, asks again. The user has two buttons, one to prompt 'Move' and one to prompt 'Stop', as in figure 6.11. This second button is to confirm that the motor seems correctly homed and is ready. When the first motor is correctly positioned, the same is asked for motor two (linear misalignment Δ) and for motor three (angular misalignment γ).
The process is not fast: the 'user-Simulink-Arduino and back' flow is slow, not a little

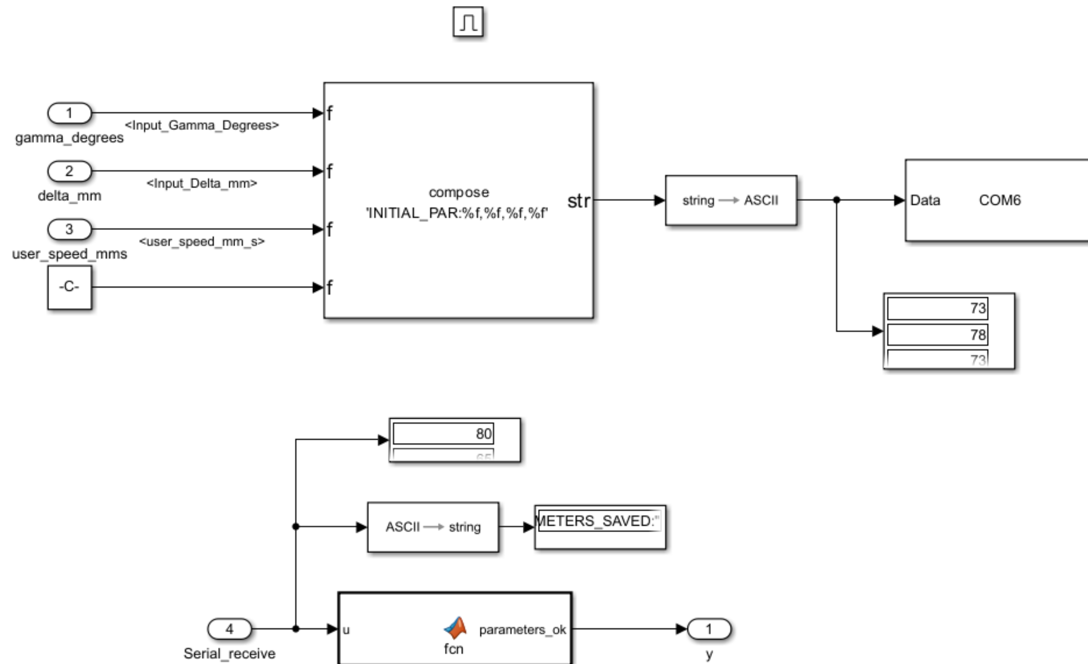

Figure 6.14: The figure shows the logic internal to the Enable subsystem, used for sending values to Arduino.

compared to a direct Arduino-user communication with Arduino IDE. However, due to serial communication constraints, this was the only option.
When all three motors are in their neutral position, the Enable subsystem (activated again by pressing 'Stop') sends as output a '1' signal, which closes the subsystem itself. For technical details: communication is performed through *Serial Send* and *Serial Receive* blocks and a MATLAB function block to manage the process.

Initialization of γ and Δ

Arduino, after completing homing sequence, uses the achieved neutral position as starting point and asks the actuators to impose the desired misalignments. The *NEMA 17* motor imposes angular misalignment γ, while *Linear Guide 2* linear misalignment Δ.
It is all done automatically by Arduino when homing is completed; *Homing* MATLAB function is responsible of receiving confirmation of correct positioning of misalignment actuators. At this point, the subsystem converts its output signal from 0 (homing and misalignment insertion incomplete) to 1: it means the prototype and the Digital Twin are initialized and ready to start the actual real-time loop.

```
### Time Span configured: 0.350 seconds ###
--- Hardware Preparation in Progress ---
Connection attempt n. 1...
Attempting to connect to Arduino on COM6...
Successful connection!
--- Hardware verified and ready. ---
--- Starting Firmware Procedures ---
Starting upload procedure
Compiling and uploading VER_1_1_1... please wait.
Upload successful! Waiting for reboot...
Attempting to connect to Arduino on COM6...
Successful connection!
ans =
     1
Starting verification for VER_1_1_1...
Sending "?" to Arduino...
Sending "?" to Arduino...
Received: VER_1_1_1
Verification successful: found VER_1_1_1!
--- Hardware and Firmware are ready! Starting simulation... ---
Saved parameters are: gamma=0.00, delta=0.00, speed=20.00, t_span=0.350
### HOMING MOTOR: 1 ###
Commands: [MOVE] to move the actuator, [STOP] when homed. Wait a couple of seconds after pushing for sync between Simulink and Arduino.
Action: Sending STOP/NEXT (n)...
### HOMING MOTOR: 2 ###
Commands: [MOVE] to move the actuator, [STOP] when homed. Wait a couple of seconds after pushing for sync between Simulink and Arduino.
```

Figure 6.15: Example of the messaged shown in the Diagnostic Viewer. Here, messages about the initialization process are readable.

Real-time loop trigger

With initialization completion, the advancement signal is set to '*1*', activating the real-time loop, which is the actual heart of the Digital Twin. Its activation pace is marked by a *pulse generator* block, that emits a square wave with period equal to the *time-step*.
The real-time control is performed by a *triggered subsystem* , activated by this pulsed wave with fixed periodicity. All the following described subsystems are indeed encapsulated inside the *Real-Time* triggered subsystem, with cyclic paced activation, one single time each new time-step.

The *time-step* length was asked in initialization phase to the user, as mentioned. Its value is constrained; if the user insert a value outside the allowed range, the closest limit value is automatically adopted for the time-step. Preliminarily, the interval was set as $[0.3 - 1]$ seconds, but the optimal values are analysed in section 7.5.
An important condition is that the prompted value is always rounded to the nearest multiple of 0.05 (seconds). The reason is the concatenation of multiple surrogate models, which simulates 0.05 seconds, as explained in the relative section.

It must be highlighted that this method of time control is possible by imposing '*Paced simulation*', with which Simulink tries to keep up with a 1:1 real-time pace. This means: a second in Simulink time is a second in clock time. Simulink *tries*, by slowing down the simulation; however, if too many computations must be performed, Simulink's time dilates.

Last thing to mention, since the subsystem has a discrete functioning, which means one iteration at each time-step. The consequence is that the outputs shall re-enter the cycle as inputs at the subsequent iteration: outputs of the jth cycle, enters at the next step as inputs. For example, x_{j+1} (*future*) becomes x_j (*present*) at the subsequent cycle; x can be any generic property. For example, considering x_{j+1} as the prediction at jth step for the next step, when this following step arrives it turns into the prediction for the

current step.
Feedback is simply performed by bringing some output signals (of the real-time triggered subsystem) back as inputs. This operation is permitted by *Unit Delay* blocks with inherited time option. It means that each time the cycle ends, the output enter the next cycle as inputs, updating their values.

Real-time subsystem: an overview

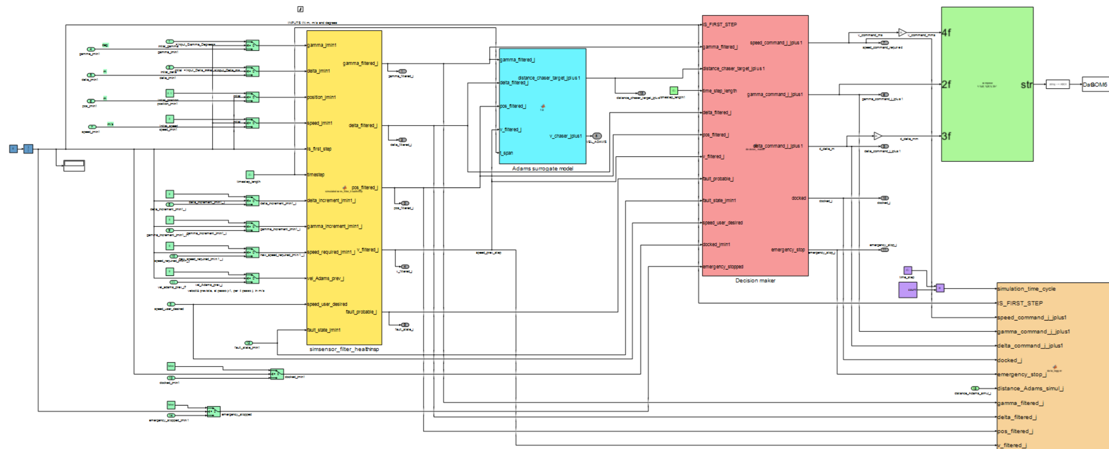

Figure 6.16: Overview of the real-time cycle subsystem (internal view). Blocks have been coloured to facilitate their description.

Figure 6.16 shows the anatomy of the real-time subsystem, which manages all the actual operations of the prototype:

- On the far left, two small blue blocks manage the initialization of this subsystem. They generate a signal indicating, with a *1* value, the first step of the cycle; this signal switches to *0* for all the following cycles.
- Many small green blocks are used to enter inputs in the subsystem. These represent the outputs of the previous cycle, but starting from initial values at the first cycle ($j = 1$).
- The yellow *MATLAB function block* includes: simulated sensor, filter and health inspector.
- Many small grey blocks outputs specific values to be used for feedback control in the following cycle.
- The light-blue block contains the Adams surrogate model, which is essential for switching from a simple control mechanism to a first draft of a Digital Twin, with predictive capabilities.

- Consequently, in red, there is the decision maker function, which bases its results on data and surrogate results.
- The resulting commands are sent to Arduino for implementation with the green blocks.
- Finally, the orange subsystem without outputs in the bottom-right angle saves data in JSON log files. Right to it, in purple, three blocks are used to keep track of the time from the beginning of the real-time phase, which is saved in the log files.

The next subsections go into detail about these blocks.

Simulated sensor

Inside the triggered subsystem there is the first main MATLAB function, yellow in figure 6.16. The option to use many distinct MATLAB function blocks would insert difficulty in managing activation and signal advancement, therefore limiting the number of MATLAB function blocks is a good option, at least in a first phase.
The first part of the function is therefore the *Simulated sensor*. It is mainly constituted by two parts: a predictive mathematical model and noise application.

The predictive part is an algebraic model that mimics the motions strategy imposed by Arduino's firmware. It uses commands and state values coming from the previous step *j-1* to predict state values for the current moment (jth time step).
For the first step, quite simply, the model is:

$$\begin{cases} p_{bn,1} = 0.1\, m \\ v_{bn,1} = 0\, m/s \\ \gamma_{bn,1} = \gamma_0, \\ \Delta_{bn,1} = \Delta_0 \end{cases}$$

First, bn stands for '*before-noise*', representing the outputs of the predictive model.
In the equations:

- p is the relative distance between the chaser and the target (magnitude value) along the track; it actually is the chaser's distance from the end point of the linear guide: these two definitions slightly differ for non-null linear misalignments Δ. It assumes the value of 0.1 meters at the beginning and reduces up to 0 meters in the contact;
- v is the speed of the chaser, in magnitude; it is null, at the first instant;
- γ assumes the user-prompted initialization value;
- Δ assumes the user-prompted initialization value.

The equations change for each subsequent step. First of all, it must be considered one thing. At the jth cycle of the triggered subsystem, the simulated sensor tries to guess, as outputs, the values at the initial instant of this jth cycle. Since the sensor is simulated, it used an algebraic model acting on data available from the previous interval, the *j-1*th.

Passing to the equations for any time-step after the first, the first concept is that the predictive model tries to follow the speed evolution carried out by Arduino and defined in the firmware. The second point is passing from the speed profile to the position; this is done by integrating velocity over the time-step.
The integration is done discretely, subdividing the time-step in 100 smaller intervals, defined as $dt_{int} = \frac{time-step}{100}$. Having j as the index of the current global time-step (in which the sun is divided), let's define k as the index of a subinterval, with $1 \leq k \leq 100$. These Subintervals are useful, in addition to a more precise integration, for considering cases in which the chaser reaches the desired speed (for the current interval j) before the interval ends, resulting in a trapezoidal speed profile inside the jth interval.
The subsequent equations follow, for any subinterval k, such that $k \cdot dt_{int} < T_{step}$:

$$\begin{cases} a_{st} = 5\, mm/s^2 \\ \\ v_{k+1} = \begin{cases} \min\left(v_k + a_s t \cdot dt_{int}, v_{req,j}\right) & \text{if } v_k \leq v_{req,j} \\ \max\left(v_k - a_s t \cdot dt_{int}, v_{req,j}\right) & \text{if } v_k > v_{req,j} \end{cases} \\ \\ a_{eff,k} = \dfrac{v_{k+1} - v_k}{dt_{int}} \\ \\ p_{k+1} = p_k - \left(v_k \cdot dt_{int} + \dfrac{1}{2} a_{eff,k} \cdot dt_{int}^2\right) \end{cases} \tag{6.2}$$

The first line is just the definition of the standard acceleration used by Arduino. It has a value of $5\, mm/s^2$ and is needed to prevent from rude accelerations. Indeed, if the speed required to the prototype at the end of the $j-1$th interval (also the beginning of jth interval, $v_{req,j}$) requires an excessive acceleration, the chaser changes its velocity according to $a_s t$ acceleration, reaching the reachable speed at the end of the $j-1$th interval. Otherwise, if the speed required at the end of the interval is not much different from the value at its beginning, Arduino asks the chaser to accelerate (or decelerate) with $\pm a_{st} = \pm 5\, mm/s^2$ rate, up to reaching this desired value; the rest of the interval is travelled at constant speed. This same approach is, as said, implemented by Arduino itself.
The second equation is actually the equation governing the speed, with the same principles just explained. The difference is just in sign, in case of acceleration (above) and deceleration (below).
The third equation computes the actual acceleration during the kth subinterval. Generally, $a_{eff,k} = \pm a_{st}$ if the subinterval sees a complete acceleration or deceleration, or $a_{eff,k} = 0$ if the subinterval is at constant speed. However, if the subinterval sees the speed reaching the desired value $v_{req,j}$, then the value of $a_{eff,k}$ would be between these two extremes. Note that, if a_{st} is always positive (it is just the magnitude), $a_{eff,k}$ considers also the sign: it is positive if the chaser is accelerating towards the target, it is negative if the chaser is decelerating (or accelerating backwards, going away from the target).
The fourth equation sees position integration. First of all, remember that the position is actually the distance between the chaser and the target (considering two points on

their respective interfaces which will adhere during the contact); therefore it has opposite sign with respect to the speed: a positive speed sees the chaser going towards the target, reducing the distance. The reference system is the reason behind a '-' sign inside the equation, both for the speed, both for the acceleration. This last equation simply implements uniformly accelerated motion over the kth subinterval.

Passing to the overall predictive model over the *j-1*th time-step, the equations are:

$$\begin{cases} v_j = v_{k=101} \\ p_j = p_{k=101} \\ \gamma_j = \gamma_{j-1} + D\gamma_{j-1\to j} \\ \Delta_j = \Delta_{j-1} + D\Delta_{j-1\to j} \end{cases} \tag{6.3}$$

For speed and position, the values are just the output of the iterative cycle over k, which started from the values at the beginning of the $j-1$th time-step to give a prediction for the current instant (the beginning of jth cycle).
γ and Δ, on the other hand, are simply computed by adding the increments ($D\gamma$ and $D\Delta$) to the values at the beginning the previous cycle. These increments were the ones computed two time-steps before (during the $j-2$th cycle), and sent to Arduino in order to be implemented at the beginning of the $j-1$th cycle. For example, during the tenth cycle the Digital Twin decides that a variation to γ is needed and sends this command to Arduino before the end of the step; Arduino reads and actuates it at the beginning of the eleventh cycle. A the beginning of the twelfth cycle, the sensor takes this increment and adds it to the value γ had at the the beginning of the previous time-step.

It must be considered that this model is far from perfect. First of all, it integrates a time-step over arbitrary 100 subintervals. Arduino's actuation cycle is a bit different. Moreover, it does not consider the time Arduino needs for other operations (communication, computations,...), which influences the motion. Indeed, the simulated sensor is just for the design and testing phase and should be replaced by actual sensors for good results. One of the main problems is not knowing whether the implementation on Arduino is actually correct, as it is not possible to obtain a quantitative feedback (only qualitative, 'by eye', which does not arrive into the Digital Twin.

After the analytic model, one point is still missing: *noise* application.

$$\begin{cases} p_{sens,j} = p_{est,j} + 0.001 \cdot \mathcal{N}(0,1) \\ v_{sens,j} = v_{est,j} + 0.0003 \cdot \mathcal{N}(0,1) \\ \gamma_{sens,j} = \gamma_{est,j} + 0.1 \cdot \mathcal{N}(0,1) \\ \Delta_{sens,j} = \Delta_{est,j} + 0.0002 \cdot \mathcal{N}(0,1) \end{cases}$$

Very simply, *Gaussian noise* is applied, by means of *randn()* command. More specific noise may be applied after considerations on the noise expected by real sensors and their behaviour.
For the noise magnitude, it was chosen an hundredth of the maximum nominal value. For the position $\frac{0.1m}{100}$; for the speed $\frac{0.3m/s}{100}$; for γ $\frac{10^o}{100}$ (approximated from 11.325^o); for Δ an indicative value $\frac{10cm}{100}$.

Considerations on sensor integration can be found sections 7.1 and 7.2.

Filtering

The sensor returns raw data that shall be cleaned as possible. This is done by a *filter*. Filter choice is a sensitive issue, which should be discussed in depth. In this work, time limited, and a simple filter was implemented without an analysis and a trade-off. A thorough analysis shall be done especially when filtering real sensors, by also evaluating their noise footprint.
Some options were anyway considered. A moving average filter was one choice, but discarded for the high delay introduced in the system. Indeed, the prototype has a dynamic behaviour which is not slow with respect to the time-step: time-steps between 0.1 to 1 second long are not negligible with respect to the run length (some more considerations about the time-step choice are in 7.5). This means that noise cleaning would introduce a worse defect by adding relevant delay.

It was then decided to use a trivial filtering method, ascribable to an alpha-beta filter, which is itself a simplification of Kalman family: the idea is to use a model estimation to weight the 'measured' value: $x_{filtered} = x_{prediction} \cdot \alpha + x_{sensor} \cdot (1 - \alpha)$, with $0 \leq \alpha \leq 1$; x is a generic variable. Two trivial cases: $\alpha = 0$ gives just the sensor output, while $\alpha = 1$ the model prediction.
The first point to discuss is which model to use for the prediction. One of the best models possible is the same one used to model the simulated sensor, which approximates the motion laws of Arduino's firmware. But, this would mean that the simulated sensor itself uses x_{pred} before applying the noise n: $x_{simsens} = x_{pred} + n$. This would involve having $x_{filtered} = x_{pred}\,\alpha + (x_{pred} + n) \cdot (1 - \alpha) = x_{pred}\,(1 + \alpha - \alpha) + n(1 - \alpha)$
Trivially, it would be sufficient to increase α: $\alpha \to 1$ would give $x_{filtered} \to x_{pred}$.
However, for the filter and the Digital Twin, the sensor is a *black box*, as it was a real sensor entering the software architecture: this argument would be a critical bias, and $x_{simsens} = x_{pred} + n$ (simulated sensor equation) in now forgotten.
For this reason, a different approach for x is here used; however, the model employed for the simulated sensor, and explained in the last pages, should be taken for better approximations to filter the signals coming from a set of real sensors in a future update.

Given these premises, it is now shown the algorithm for the filter.
The first point is the computation of the speed (magnitude value) of the chaser, with a predictive model. For this phase, it was used:

$$v_{\text{pred},j} = \frac{v^{\text{prev}}_{\text{Adams},j} + v_{bef.noise,j}}{2}$$

This formula exploits the available predictive tools, with an average between these two values:

- $v^{\text{prev}}_{\text{Adams},j}$ is the output of the simulation made the step before (*j-1*), of duration *j*. It uses the surrogate model based on Adams multi-body model, as described in the relative section.

- $v_{bef\,noise,j}$ is the speed as predicted by the same model used for the simulated sensor itself, which simply is the speed value inside the simulated sensor before noise addition.

$$p_{\text{pred},j} = p_{j-1} - \frac{T_s}{2}\left(v_{j-1} + \frac{v^{\text{prev}}_{\text{Adams},j} + v_{bef.noise,j}}{2}\right)$$

The predicted value for chaser-target distance is obtained, in Adams, with a strong approximation. It indeed assumes motion with a linear speed profile. This equation is derived by integrating over the subinterval jth of a linear speed profile:

$$\int_{p_j}^{p_{j+1}} dp = \int_0^{dt_{int}} -\left[v_j + (v_{j+1} - v_j)\frac{t}{T_s}\right] dt$$

T_s is the length of the time-step.
Solving the integral, it is obtained $p_{j+1} = p_j - \frac{T_s}{2}[v_j + v_{j+1}]$ This equation simply considers the trapezoid area as the increment, since the speed profile is linear.

For γ and Δ the filter is trivial, as earlier explained: the predicted values are the same used for the simulated sensor:

$$\gamma_{\text{pred},j} = \gamma_{j-1} + \Delta\gamma_{j-1\to j}$$

$$\delta_{\text{pred},j} = \delta_{j-1} + \Delta\delta_{j-1\to j}$$

A first-order exponential (alpha) filter is applied to combine prediction and measurement:

$$\begin{cases} v_{f,j} = \alpha\, v_{\text{pred},j} + (1-\alpha)\, v_{\text{sens},j} \\ p_{f,j} = \alpha\, p_{\text{pred},j} + (1-\alpha)\, p_{\text{sens},j} \\ \gamma_{f,j} = \alpha\, \gamma_{\text{pred},j} + (1-\alpha)\, \gamma_{\text{sens},j} \\ \Delta_{f,j} = \alpha\, \Delta_{\text{pred},j} + (1-\alpha)\, \Delta_{\text{sens},j} \end{cases} \qquad \alpha \in [0,1].$$

The filter is a simple first-order one. It was firstly used adopted $\alpha = 0.3$, but with calibration the value will be changed into 0.1.
As mentioned, this simple filter should be refined, especially after the integration of a real sensor: this is a simple version, useful for testing and not yet for effective control.

Health inspector

Just after filtering, it is the *health inspector* module, which tries to signal immediately possible faults. This allows a shortcut to stop the run, before finishing it, for prevention reasons; the stopping command is elaborated in the *decision maker* module.
The block is integrated in the same MATLAB function block of sensor and filter, but can be easily separated. It includes, at the moment, simple logical and physical checks, that can be extensively extended with many other proofs. Up to now it is not a robust health-status verification, but just an attempt to evaluate the functioning of the system. Many other conditions may be imposed to identify faults on the sensors, on Arduino, and on other components of the cyber-physical system.
Moreover, the significance of this module is now limited: its usefulness will be enhanced by actual sensors integration.

In the current version, the checks can be grouped into:

- *Velocity integrity*: to identify negative values of speed (meaning backwards, moving away from the target). It is not actually a fault, but an undesired behaviour, associable to possible problems. To account for sensor's noise, a threshold is set: -0.003 m/s (instead of less than exactly 0.000 m/s, to avoid false alarms due to sensor inaccuracies).
  Moreover, to protect from uncontrolled over-speeding, the condition of not exceeding 110% of the user-defined maximum desired speed.

- *Consistency of misalignment actuators*: since the Digital Twin can send, each cycle, corrections for γ and Δ, here there is a check on their implementation. The module verifies if the misalignment values are correctly changing, increasing or decreasing, concordant with the sign of the commands; the module checks for signal inversion (in the signal propagation towards the actuators) and for possible motor locking. The formulation is, again, very simple, and can be refined in future updates.

When one of these conditions is recognised, a *fault* flag activates. To avoid false alarms, it is effective only if it activates for two consecutive cycles: this double check is performed, later, in the *Decision maker* module.

Again, note that many other checks could be implemented in this phase, in order to ensure solid security to the operations, at different levels (software, actuators, logic,...). An example is checking if the chaser has stopped before arriving at docking contact. The purpose of this work is indeed to build the initial software platform, with infinite possible improvements, especially with a full hardware integration.

Surrogate model integration

An important module is the *surrogate model*, light-blue in figure 6.16.
As seen some pages ago, three 5-dimensional (4 inputs to one output each) *response surfaces* were defined. Each one is represented by a quadratic polynomial, as already described. The three formulas belong to the same model, obtained by fitting the same set of simulations (as described in the subsection about *outliers cleansing*). Some considerations about the accuracy of the three polynomials have already been done in subsection 6.3.2 and more is said in section 7.4.
To resume, the three polynomials are:

- $v_{end}\left(\gamma, \Delta, p_0, v_0\right)$ approximates the velocity of the chaser at the end of the 0.05 seconds simulation.

- $p_{end}\left(\gamma, \Delta, p_0, v_0\right)$ approximates the position of the target on its Linear Guide, with a value of 0 m at the end (when docked, at contact with the target) and 0.1 m at the maximum distance, at the end of the 0.05 seconds simulation (the multi-body model was taken in runs of 0.05 seconds to build the surrogate).

- $a_{target,RMS}\left(\gamma, \Delta, p_0, v_0\right)$ approximates the acceleration of the target interface, which is connected to the structure by four springs and sustained by a circular annular

sustain (made up of two halves, both held in place by a spring, as figure 6.7). This approximation is however very poor, and not directly useful for the control cycle. It is therefore not included in the Digital Twin for this preliminary version; future updates are reserved to refine the model, possibly introducing a better estimation of the dynamic actions acting on the target's interface. Only the first two polynomials are thus considered.

Adams Insight allows to export the model in *.m* format, for MATLAB; small edits were needed to convert it in the right form. The formulas are, originally:

$$y_k = c_0 + c_1 u_1 + c_2 u_2 + ... + c_n u_n + c_{1,2} u_1 u_2 + c_{1,3} u_1 u_3 + ... + c_{n-1,n} u_{n-1} u_n + c_{1,1} u_1^2 + ... c_{n,n} u_n^2$$

generalized as:

$$y_k = c_0 + \sum_{j=1}^{n} c_i u_i + \sum_{j=1}^{n} \sum_{h=j+1}^{n} c_{ij} u_i u_j + \sum_{j=1}^{n} c_{ii} u_i^2$$

In the MATLAB function for the surrogate model, it is firstly defined the matrix of the exponents E, needed to build the second degree polynomial:

$$\mathbf{E} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \qquad \mathbf{u}_k = \begin{bmatrix} \Delta \\ \gamma \\ v_0 \\ p_0 \end{bmatrix}$$

u_k is the input vector, with the values at the beginning of the simulation. From these two arrays, it is computed the *basis vector* ϕ.

$$\phi(\mathbf{u}_k) = \begin{bmatrix} \phi_1(\mathbf{u}_k) \\ \phi_2(\mathbf{u}_k) \\ \vdots \\ \phi_{15}(\mathbf{u}_k) \end{bmatrix} = \begin{bmatrix} 1 \\ \Delta \\ \gamma \\ \vdots \\ v_0^2 \\ p_0^2 \end{bmatrix}$$

Each term is obtained by $\phi_i(\mathbf{u}_k) = \prod_{l=1}^{4} (u_{k,l})^{E_{i,l}}$.
For some clarity: i is the index of elements of ϕ ($1 \leq i \leq 15$); j is the current cycle of

the Digital Twin real-time run; k is the sub-cycle, defined a little further on (not to be confused with k of the simulator sensor); l is the index of u_k $(1 \leq l \leq 4)$.
Adams Insight gives the list of the coefficients of the polynomials, ergo:

$$\mathbf{c}_{dist} = [c_{p,1}, c_{p,2}, \ldots, c_{p,15}]^T \in \mathbb{R}^{15}$$

$$\mathbf{c}_{speed} = [c_{v,1}, c_{v,2}, \ldots, c_{v,15}]^T \in \mathbb{R}^{15}$$

Simply, to obtain the approximation of p and v at the end of step, *k+1*, which is 0.05 seconds later, it is possible to write:

$$p_{k+1} = \mathbf{c}_{dist} \cdot \boldsymbol{\phi}(\mathbf{u}_k) = \mathbf{c}_{dist}^T \boldsymbol{\phi}(\mathbf{u}_k)$$

$$v_{k+1} = \mathbf{c}_{speed} \cdot \boldsymbol{\phi}(\mathbf{u}_k) = \mathbf{c}_{speed}^T \boldsymbol{\phi}(\mathbf{u}_k)$$

Since the Digital Twin can accept different length cycles (constant however during the whole run), simulation time should be another input for the surrogate (which is saying 'simulate what happens for T_{step} seconds', with T_{step} user-defined). This approach is complicated to be translated in the creation of a surrogate with time as input variable (that is, inside u_k).
A different approach was so chosen, at least for this preliminary version of the software. The surrogate simulates for 0.05 seconds; the user can choose the time-step (provided that within a certain range), which is however rounded at the closest multiple of 0.05, rounded up if halfway. Therefore, each jth interval can be decomposed in k sub-intervals of 0.05 seconds length.
At each jth cycle, the surrogate model algorithm is:

$$\text{for} \quad k = 1, ..., n \quad \text{with } n = \frac{T_{step}}{0.05}: \quad \begin{cases} \boldsymbol{\phi}(\mathbf{u}_k)_i = \prod_{l=1}^{4} (x_{k,l})^{E_{i,l}} & \text{for } i = 1, \ldots, 15 \\ v_{k+1} = \mathbf{c}_{speed}^T \boldsymbol{\phi}(\mathbf{x}_k) \\ d_{k+1} = \min\left(\mathbf{c}_{dist}^T \boldsymbol{\phi}(\mathbf{x}_k),\, 0.1\right) \end{cases}$$

At the end of the cycle over k:

$$d_{j+1} = d_{n+1}$$

$$v_{j+1} = v_{n+1}$$

What happens is that for all the internal 0.05 seconds sub-cycles, the output values of p and v re-enter the polynomial for another 0.05 seconds estimate. The ending values, the prediction for the next step *j+1*, are the output of the last iteration over k.
Note that, to roughly correct errors of the surrogate, when the output p value is greater than *0.1 m*, not physically consistent (the actuator cannot extent itself beyond its limit switch), are truncated to 0.1 m. Moreover, being inside the big jth cycle, γ and Δ are considered invariable, since they may change at the beginning of each cycle due to DT commands; the implementation of misalignment corrections is approximated as instantaneous.

Note a strong approximation, originating from the original multi-body model: the surrogate model considers motion at constant speed, which is not realistic, since the chaser accelerates trying to catch a goal speed at each step.

A future refinement is needed. The new model should have two inputs velocities: one, the initial velocity at t_k; the other, the expected velocity at t_{k+1} or, alternatively, the expected acceleration. The multi-body model should so simulate accelerated motion. The surrogate model in the Digital Twin should take the time step j, subdivide it in k sub-intervals of fixed length (such as 0.05 seconds) and implement the formula for this accelerated motion. Different approaches can be considered, from constant acceleration to a refinement of the predicted speed at each sub-step k before using it as value for $k+1$ estimation.

Another big issue is that the output is always positive, since the surrogate always returns positive values, since they were defined as the magnitude of p_{end} and v_{end}. During the nominal phase, with the chaser approaching the target, speed is positive, and thus no problems are encountered.
However, if the speed reverse its sign, for example with the chaser arrived at limit switch with non-null velocity and returning back, a problem arises: the kth output speed should be negative, but it is returned in magnitude and is positive. In the initial conditions for $k+1$ sub-cycle prediction, speed is positive instead of negative, that would instead be correct. This inconsistency starts a numerical loop, not representative of reality.
This problem, again, is known and shall be solved in potential future updates, with the revision of the simulator.

Finally, a note on the block architecture. The surrogate model is a MATLAB function block inside the real-time subsystem. Since the syntax can be compiled in *C* by Simulink, it does not need to use extrinsic function recalls in MATLAB.
The function simply accepts γ_j, Δ_j, p_j, v_j and T_{step}. These values arrive, filtered, from the (simulated) sensor. The function returns, as outputs, p_{j+1} and v_{j+1}, predictions for the next time-step.

The decision maker

The *decision maker* module is the core of the control strategy of this specific Digital Twin. It should be noted that the DT may be parallel to the actual control system of the system; in the current architecture, however, they are integrated. The *decision maker* is visible in red in figure 6.16. It is, again, a MATLAB function block, which analyses filtered sensor data and the predictions of the surrogate model to determine the commands to send to Arduino.
First of all, two flag values are verified:

- *docked* is a boolean signal turning *true* when the chaser reaches the theoretical docking position. A chaser-target distance value of 0.002 m is taken as threshold, to withstand possible noise fluctuations (instead of a theoretical value of $p=0\ mm$);

- *emergency_stop* is a boolean signal accounting for the possible presence of faults; it turns *true* when two consecutive cycles (*j-1* and *j*) show the possible presence of faults (*health inspector*).

Both these conditions stops the prototype, imposing null velocity. For both of these checks, a more accurate version should be implemented, for higher security. Also, the presence of impact sensors would require different and more effective checks.

A message is printed, for the user, when these conditions turn *true*. Additional messages could be shown to the user, but it was avoided not to slow down the real-time pace (obligating to use wider time-steps). Better strategies to communicate the prototype state during the simulation can be developed, such as printing periodically important values after a certain number of cycles. Also, a refined health state supervision could offer the possibility for other useful messages to print. For monitoring purposes, the user can add *Display* or *Scope* blocks inside the Simulink model.

For the *speed profile*, the following control laws are imposed:

- *Cruise phase*: if the chaser appears not to be close to the target ($3\,cm \leq p \leq 10\,cm$, which is the maximum elongation of the linear guide), the user-defined cruise speed is imposed as goal velocity. The chaser accelerates or decelerates following Arduino's commands (see later for more details).
- *Negative speed protection*: if the velocity appears to be negative, with the chaser moving away, a slightly positive value is imposed (2 mm/s).
- *Proximity deceleration*: when the chaser approaches the target ($p \leq 5\,cm$), speed is reduced of 10% each cycle ($v_{j+1} = 0.9\,v_j$), to avoid strong impacts. This law is arbitrary and should be optimized in the following refinements, calibrating with tests. Also, to prevent the chaser from stopping too early, a base value of 2 mm/s is granted anyway (verifications should be done on the minimum value that can be smoothly managed by the actuator).

As said, the chaser stops imposing null velocity when docked.

For *misalignment control*, discrete commands are sent with the goal of bringing their values inside the requirements range (see 4.1):

- If γ value exceeds $\pm 5^o$, a $\mp 1^o$ correction is sent each cycle, until the angular misalignment is limited inside the range.
- Analogously, if Δ value exceeds $\pm 10\,mm$, a $\mp 1\,mm$ correction is sent each cycle, until the linear misalignment is easily bearable.

These commands (the goal speed for the velocity and the misalignments corrections) are sent to Arduino, which actuates them. See section 6.3.4 for details.

Clearly, these control laws, combined with the algorithm Arduino uses to actuate them, could be improved. Possible refinements can include greater uniformity in the speed profile and more precise corrections for γ and Δ. Any update however will start from a stable and working base, which is the goal of this project.

Sending commands to Arduino

Arduino looks periodically if there are commands incoming from Simulink. To summarize, the commands are:

- The goal speed value to aim a t_{step} later; Arduino tries to reach it gradually (see later for more details);

- The possible increase or decrease of angular misalignment γ;
- The possible increase or decrease of linear misalignment Δ.

In Simulink, a *Compose String* block packs all the three values with syntax `<%4f,%2f,%3f>`, in which the three commands are in the same order of the list above, in place of the `%` with the specified number of digits. Then, after a conversion into ASCII format, the message is simply sent to Arduino by a *Serial Send* block. See figure 6.16 for details.
Note that in this communication mode between Simulink and Arduino there is no feedback as instead was done when sending initial parameters: here, Arduino does not send back a confirmation. The feedback will be given, in a second iteration of the Digital Twin, by the sensors.

Data logging

As seen during the functional analysis, storage is needed to keep track of data. While the current version of the controller only uses values of the last one or two cycles for computations and predictions, with internal Simulink memory for it, future updates may exploit data archive for optimization, monitoring, filtering and predicting algorithms. Moreover, already at present, data are useful from verification and calibration activities to data analysis.

The module doing this is the *data logger*, orange in figure 6.16. It saves an external JSON file (.jsonl to be exact) with a set of important parameters, logging once for each control cycle. The file is stored in a specific folder, inside the global folder of the DT.
The module is a MATLAB function block that acts as interface with an external MATLAB function (saved as *.m* file in the Digital Twin folder), since some of the used commands cannot be directly compiled due to Simulink limitations: it is therefore necessary to use the extrinsic mode.

Each new run, a new file is created, with the name reporting information about it: *'Log_yyyy-mm-dd_HH-MM-SS.jsonl'*. This is performed by using the *IS_FIRST_STEP* signal: it has *1* value at the first cycle and then immediately switches to *0*. It is used many times in the middleware, generally for values initialization purposes.
For all the following cycles, this file is kept open, with a *Persistent File Identifier* (which keeps the reference to the model persistent as long as necessary); this trick allows to speed up the compilation, without continuously opening and closing the file. So, during each cycle, MATLAB appends a new line with important parameters for the current cycle. Limited numbers of digits are imposed to speed up data logging.
For better data consistency, scientific notation would be preferred to the currently used extended formulation, giving uniformity to the precision of variables having different orders of magnitude. In the first upgrade, this can be implemented.

Here is an example of the first line of a file:

```
"time":0, "speed_cmd":0.02, "gamma_cmd":-1, "delta_cmd":0, "docked":false,
"fault_stop":false, "dist_Adams_predict":-1, "gamma_filt":7.841881,
"delta_filt":0.00506, "pos_filt":0.100376, "speed_prev":0.000385
```

Variables are:

- *time* is the time from the beginning of the autonomous real-time run (meaning it does not count the time used for initialization phase); it is tracked by a *Counter* block which counts the number of cycles and multiplies it for the time-step value;
- *speed_cmd* is the command sent to Arduino for the goal speed (*Linear Guide 1*), to be reached the next time-step (*j+1*);
- *gamma_cmd* is the command sent to Arduino for the angular misalignment γ; it is an increment ($\pm 1^o$), to be applied the next time-step (*j+1*);
- *delta_cmd* is the command sent to Arduino for the linear misalignment Δ; it is an increment ($\pm 1\,mm$), to be applied the next time-step (*j+1*);
- *docked* is a boolean variable that signals the reaching of theoretical docking position;
- *fault_stop*: is a boolean variable that reports fault state, output of the *health inspector* module;
- *dist_Adams_predict* is the chaser-target distance p predicted at the step before by the surrogate model for the beginning of the current time-step. The example shows *-1* value since it is the initial cycle: no simulation has already taken place, therefore *-1* is just a placeholder value, without a physical meaning;
- *gamma_filt* is the value of γ as output of sensor's filtering;
- *delta_filt* is the value of Δ as output of sensor's filtering;
- *pos_filt* is the value of chaser-target distance p as output of sensor's filtering;
- *speed_prev* is the magnitude of chaser speed, as output of sensor's filtering;

All values are reported in IS units.
The file is ready for reading after the simulation.

Termination of the simulation

The run of Simulink is built to be terminated when the Digital Twin identifies a possible fault or considers docking successfully achieved. It is possible using a *Stop simulation* block, activated if *docked* or *emergency_stop* switches to *true*, as seen in figure 6.12. Also, messages are appended in the Diagnostic Viewer, to inform the user, also specifying which of the two reason is behind the ending of the simulation.

6.3.4 Arduino firmware

Here is briefly described Arduino's firmware, whose code is in *Appendix A* (11). Please note that many comments are in Italian: the code was not refined for the reader.

In the preamble, parameters are defined or initialized. Pin definition follows, ensuring Arduino is aware of the hardware layout of the circuit. Note that that microstepping is not enable.

Then, a long part is just inherited by the code written by Binetti for prototype's control in his work.[1] In particular, helpful functions are defined, even if not all of them are employed in the actual loop.
The setup assigns serial communication rate, imposed to 115200 for higher speed, and pins.

Finally, the loop starts: it uses many boolean parameters to know which part of the loop activate in any iteration. These boolean parameters are generally *false* up to the action is complete, and *true* to switch to the subsequent section.
These sections are:

- *Parameters reception*: this part receives from Simulink and updates some parameters, such as the initial misalignments and t_{span}; then it sends back confirmation.
- *Homing*: the second section is responsible of executing user commands for homing, by sending the proper impulses to the three A4988 drivers, which then operates the actuators.
- *Misalignment initialization*: this section autonomously imposes γ_i and Δ_i, then sends the conformation to Simulink.
- *Real-time loop*: this part is the final control part for docking, equivalent to the real-time cycle triggered subsystem in Simulink. At each cycle Arduino checks for update values from Simulink, then actuates possible variations for misalignments Δ and γ (respectively, motors 2 and 3).
  It is then computed and actuated the ramp for chaser's speed for approaching motion.

The motion ramp is computed trying to align the speed with the actual speed desired by Arduino. Arduino uses a small cycle for these iterations (motion of about 0.01 seconds), meaning that the motion is done with very small steps at constant speed. This first compromise between precision (small steps to approximate a linear ramp) and continuity (steps not so small to ensure a continuous movement) is probably still not good to produce a satisfactory continuous motion.
A profound refinement of the motion strategy is probably needed to optimize the code, trying to get the most smooth movement for the actuator. One change that could be done is using the same time step of Simulink (instead tiny 10 ms steps), during which the motion can be continuous without interruptions. However, this would mean checking for parameters update only every 0.4-1 seconds. This could lead however to two criticalities. The first is that the two cycles can be out of phase, with Arduino delayed. The delay can be as long as almost the time-step length, which is a considerable amount of time.
The second point is that the cycle is not only occupied by the chaser's motion: other logical steps (such as skipping the *if* cycle for homing) and operations, to compute the motion commands for the actuator, can subtract time. The risk is thus to have a longer cycle than wanted. Since the motion would last for t_{step}, the cycle would globally last a bit longer:

$$t_{actual} = t_{logical\,operations} + t_{step} > t_{step}$$

This should be avoided with specific time management strategies.

The current firmware configuration is just preliminary. Here, the goal was to test the architecture of the Digital Twin, and time constraints penalised the construction of a solid firmware code. It needs rework to be fully functional, net of Arduino's limitations (see section 8.2 for details). Anyway, modifications to the firmware do not require middleware modifications, so that there is a certain degree of independence between the two.

6.4 Simulation instructions

Here is a brief section, to use as support for a user starting a run of the Digital Twin. First of all, all the needed files should be in the same folder. In particular, there should be Simulink's model (the middleware), a folder with Arduino's firmware, some MATLAB functions, a folder for log files. The first step is to open the file `Digital_Twin_docking.slx`; it is mandatory that MATLAB *Files* window is open on the main folder `Digital_Twin`. Then it is possible to start the simulation on Simulink, by clicking on the *Run* button, at right in figure 6.17. Note the small gauge icon under the *Run* green symbol: it means 1:1 pace, that must be active to slow the pace down to real-time.

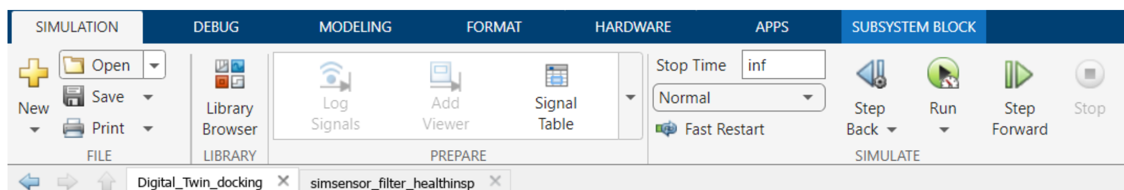

Figure 6.17: Simulink commands for starting a new run.

Just after the start, before Simulink compiles, it asks for parameters in input. First, misalignment and cruise speed (fig.6.13); then, the time-span (fig.6.18) and the version identifier code for the firmware (fig.6.19).

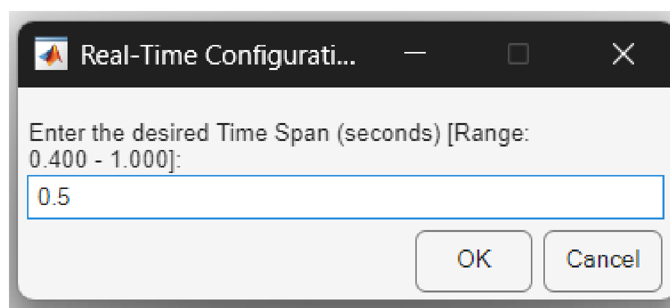

Figure 6.18: Window asking to enter the time-step.

At this point, as the DT tries to connect with Arduino, in case of problems it may ask to check if the USB cable is actually connected, if other programs are holding Arduino's communication (such as *Arduino IDE*), or if Arduino is recognised at port *COM6*. It may be necessary to open the computer's *Device Manager* to change the *COM#* of Arduino

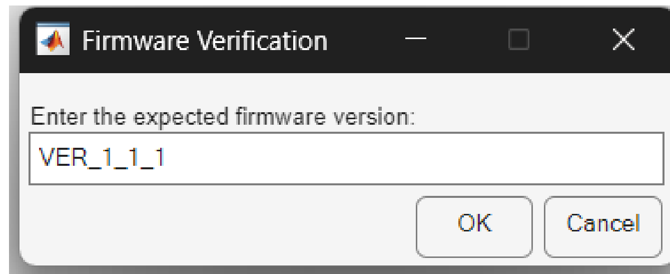

Figure 6.19: Window asking to enter the firmware identifier code.

into *COM6*. Unfortunately, there is no flexibility in the port number and changing it in the *Device Manager* is faster than changing it in the Simulink model.
As already seen, another critical point is the *Serial Configuration Block*, which rarely it does not work; in section 6.3 it is explained how to replace it in case of problems.

After the pre-configuration part, Simulink compiles the model and starts it. During the remaining initialization phase, also *homing* is performed: for it, the user shall act on two buttons, as seen (figure 6.11).
Then, the real-time starts autonomously up to the end of the simulation. Each step is followed by messages in the *Diagnostic window*, as in figure 6.15. At the end, in the `Digital_Twin` folder the user finds the sub-folder *DATA_LOGGER* with the log file.

6.5 Software verification

Approximately following the guidelines in section 3.1.4, with natural variations due to the specific features of this projects, it is worth mentioning the *verification* process.
As software grows piece by piece, it is important to check if it works in the right way. Later, it will time to check if it is the right product, meaning that it performs the functions for which it was conceived; this goes in the final stages of the thesis, with *validation* and *testing campaign*. While *validation* is more quantitative and focused on the results of the product, *verification*, by contrast, is more qualitative in nature, with the aim of ensuring that results are achieved. Anyway, both must still receive adequate attention.

Considering the multi-body model as a separate previous unit, a work plan was built for the Digital Twin and the development of the surrogate model. It condensed the Digital twin architecture and its functional analysis into a block structure that could cover all the needed aspects of the DT. Specifically, this scheme was hand-written and iteratively adjusted, so it is not worthy to show it, but the overview in section 6.3.3 is the close to it. What remains of this block structure in this essay is the subdivision in blocks of the DT description (section 6.3.3).
This decomposition of the work strongly simplified logical and temporal organisation of the work, but had also a second benefit: it allowed to verify each segment by itself, instead of the whole middleware once at the end.

Therefore, *verification* progress has been progressive and spontaneous, by considering each functional block at a time, during the programming work. This mainly means debugging, and verifying that each block satisfies its functions.

The process was facilitated by a secondary Arduino board, not connected to the actuators and to the prototype, just to debug the code during the development. Naturally, some blocks were considered verified, but when moving on to other new sections, it became necessary to go back and correct, refine or redo some other middleware units.

For *verification*, it is important not to only consider mere debugging, but also to simulate various scenarios, including anomalies and off-nominal. Rules of good coding are almost sufficient for a solid verification, taking for granted that problems could emerge in following tests, and often during the operative life. However, some attention must also be paid to individuate and satisfy the functions each block is destined to, a sort of low-level validation of the various subsystems.
Note that this work covers the first iteration of the project. In case of advanced iterations, *verification* and *validation* shall receive more attention, and perhaps *verification* should include a specific testing campaign for overall integrity, with nominal and off-nominal scenarios. Moreover, any must undergo regulatory campaigns before actual Space usage.

6.6 Additional algorithms

The Digital Twin's architecture was kept simple, as outlined in the preceding sections. Thus, no other algorithm was added for better performance, since the Digital Twin is still far from being fully operational. The inclusion of additional tools is intended to make the DT smart and more autonomous. However, this work is much more preliminary than this, representing only the initial iteration of the design process.
First, it is important to have a fully operational product, with solid control algorithms. Then, developers can work on adding more layers of the pyramid in figure 4.8. Currently, this work goes as far as the third layer (prediction capability), with all its limitations.

Complex algorithms can be implemented in an advanced refinements. *Machine learning* learns from past simulations to refine the control strategy and the simulator. Similarly, the model can also be improved by strategies using *Predictive Graphical Models (PGM)*. Also, *AI* is spreading rapidly and can offer similar benefits.
Currently, the simulator only predicts the state after one t_{step}. However, it could be expanded to predict the evolution of the whole run during each step (if computational power is available) for optimization purposes. Optimization is not a primary design goal, coming after the definition of a solid operative architecture.

Chapter 7

Implementation of the Digital Twin

7.1 Sensors architecture

A section should be devoted to the description of the *sensors architecture*. From the hardware point of view, the following should be clarified: how sensors are placed in the physical system; how they exchange data, possibly with quantitative budgets; what is the precision of these data.
Also, software observations should consider: if data are sent to the Digital Twin raw or already processed; which operations are indeed performed already inside the sensor, before transmitting data; rates of sampling and data sending; protocols and format used for data exchange; noise characterisation.

Perhaps the most important point is about data, first of all on how to extract the desired information from data provided by the sensors. It is also important to consider how actuators and sensors are related in the feedback; however, this subject should have already been discussed during the preliminary analysis and the choice of sensors.
Finally, preliminary considerations should also be made regarding the response to possible sensor failures.

For this work, there is not much to say, since only a simulated sensor was employed to preliminary test the Digital Twin architecture. Successive iterations introducing actual sensors will provide more detail on the architecture.

In the current situation, the *simulated sensor* is simply a MATLAB function inside the Simulink modelled middleware, as seen. It takes data from the previous step and uses it to extract information about the current instant, then adding numerical noise in the process. The same MATLAB function also hosts the filter.
A first, simple, evolution would be to isolate the sensor in an individual MATLAB function block. This way, the filter would be independent, in terms of architecture, seeing the sensor as a black box. This would pave the way for the integration of physical sensors. However, compared to the current configuration, no substantial changes would happen.

7.2 Sensors integration

Speaking about the integration of the sensors in the prototype and in the Digital Twin, virtualization should be considered as an option (section 6.1) should be made. Furthermore, a description of the integration process, in details, should be made at this point. It should include, in addition to the possible virtualisation, a review of filtering and elaboration module. Please note that the latter is the module that takes incoming raw data to convert in the format readable by the Digital Twin. This block could be host in the middleware or even outside Simulink environment; it could or even be delocalized, from an hardware perspective, into the sensor itself. Currently, again having a simulated sensor, these considerations cannot be made.

Moreover, some space could be left for validation of the simulated sensor, which has not been done here. However, without non-embedded sensors available, validation of the current strategy can not be made. The discussion could be resumed after the final tests, comparing the sensor's predictions with the actual behaviour of the prototype. As there is no time in this work, all is left to future work.

7.3 Integration of the Digital Twin

This section briefly discusses how to integrate simulators and prototypes into digital twins; or rather, how digital twins fit into the overall architecture.

7.3.1 Introduction to co-simulations

In this work, a surrogate model has been used for simulations. This is often the right path, since complete models require more power than the available. However, a brief excursus about *co-simulations* is presented, with the idea of being helpful for possible future developments.

Complex systems require complex simulators. Developments in electronics allow to reach high computing power, which is crucial for fast and composite simulations. However, in addition to hardware constraints, other problems should be addressed[73]:

- Different models are built on different platforms, sometimes commercially available ones requiring specific licences and with Intellectual Property;
- Distinct models may require different computational resources and different computational speeds;
- Inputs and outputs of different simulations may use different data formats;
- Simulations may be required to be real-time or faster, sometimes integrated with external sensory data, and with additional algorithms; this is always true for Digital Twins;
- Not all solvers treat simulated time at the same way.

These sparse considerations naturally hold also for Digital Twins (the fourth point is almost a DT definition), recalling the management of co-simulation. A *co-simulation* consists of all tools, theoretical and practical, necessary to integrate distinct simulators in a global simulation, for holistically simulate the whole system. For the sake, each individual simulator is intended as a black box, which acquires inputs and returns an output.[73] Thus, a *controller*, sometimes called *orchestrator* or *master*, is needed to combine them and manage queries and consequentialities, acting at a high-level and controlling time progression.
A distinction on model properties is needed. When talking about a model describing a physical system, or a phenomenon, reference is made to *validity* as the difference of the result of the model and the real behaviour. Secondly, the property of a simulator well-fitting the model is *accuracy*. This distinction is clear in the dissertation of the surrogate model (sections 6.3.2 and 7.4).
Each individual simulator must satisfy good levels of validity and accuracy, as well as then the global co-simulation must (now treating each simulator as a black box). Accuracy may be esteemed by considering the distance between the solver results and the analytical model ones, from which the simulator was derived. The analytical models are however often not available; then, other strategies has to be adopted, which are later outlined.
Another concept to be clarified is time. *Simulated time t* (sometimes only *time*) is the time flowing in the simulated environment. Instead, *wall-clock time τ* is the time passing in the real world during the simulation, being the time required by the machine for the computations. Their relationship is $t = \alpha\,\tau$, with $\alpha = 1$ meaning a real-time simulation. For a Digital Twin, to allow predictions, simulations must be faster than the reality, resulting in $\alpha > 1$.

Discrete Event, Continuous Time and Hybrid Co-simulations

It is necessary to discern between three categories of co-simulations. [73]
Discrete Event co-simulations represents phenomena with instantaneous reactions to external stimuli. Moreover, more than one changes and events may happen in a single moment, with the orchestrator that has to account for it; if there is a specific causal order for distinct simultaneous occurrences, it is said *super-dense time*. Thus, instead of continuous quantities, the focus is on events: each one is elaborated while simulated time stops for an instant.
For distributed co-simulations, it is often adopted the IEEE standard *High Level Architecture (HLA)* for the integration of *Discrete Event* simulations.

On the contrary, *Continuous Time* co-simulations handle time-continuous domains. It is thus defined a *communication step size H* (or *macro-step size*), which is the time interval between two forthcoming synchronisations. The orchestrator should indeed manage the different micro-step sizes h of the diverse simulators.
It is possible that the individual solvers have h smaller than H, often a strategy to lighten computations, with less global updates than the computations. In this case, however, between two global communications with the orchestrator, simulators need to formulate hypotheses for replacing missing inputs from other simulators; various extrapolation techniques may be adopted, such as linear, polynomial, extrapolated-interpolation, or

advanced methods such as context-aware extrapolation. Constant extrapolation may be dangerous for creating non-continuous step-functions. Furthermore, for precise co-simulations, after each new time-step H refreshing, it may be needed to return back to the previous steps to correct inaccuracies caused by extrapolations. Stored-data should then be accessed and a retrospective interpolation performed; this process however is a stall in time progress and may be detrimental to real-time.
This is the *Jacobian approach*: all the simulators advance together from t_i to $t_i + H$ simultaneously, with the orchestrator storing the outputs in a proper sequence at each time-step and using them as the inputs for the next time-step.
A different approach is however the sequential *Gauss-Seidel* one, which sees the simulations proceed one at a time to the next step, gradually using the data already collected by the simulators that have already moved on. A simulator S_1 advances from t_i to $t_i + H$; then, a second solver S_2 does the same, but using as inputs all the extrapolated inputs but for the ones coming from the first simulator S_1, for which an interpolation approach is possible. Thus, simulator S_j uses interpolations for the inputs coming from $S_{1,\ldots,j-1}$ and extrapolations for the inputs coming from simulators not yet advances. When all the simulators have reached $t_i + H$, the simulations continues the same way for the next interval. This technique may be more accurate thank to interpolation, but does not allow parallel computations.
An important tool for Continuous Time co-simulations is *FMI (Functional Mock-up Interface) standard*, an interface for simply combining different simulations. [74]
Regarding accuracy, it has be seen that rarely the exact analytical solution of the model (which validity has to be evaluated separately) is available for error estimation. [73] For single simulations, it is generally true that minimizing the micro-step size $h \to 0$ brings to a decent convergence. For co-simulations it is not always true that decreasing the macro-size step H gives the same advantages, because of possible coupling effects. Many factors other than H are involved in accuracy: the models themselves, the simulators, the extrapolation methods, the overall time-span to be simulated. However, if each single simulator is convergent and the coupling method is such that the global model can be written in state space form (see [73] for more details), the co-simulation is also considered convergent. This property enables to use various techniques for error estimation, such as Multi-Order Input Extrapolation and Parallel Embedded Method.
Error estimation may lead to corrective actions such as adapting H in the next steps or implementing recursive corrections.

Hybrid co-simulations, in the end, combine the feature of the previous two types, integrating systems with continuous features with others with discrete behaviours (on/off states). A short example deserving a mention is when a system changes its state as a discrete response of a continuous signal exceeding a threshold value, such as an air conditioner switching on for high temperatures. Temperature is indeed a continuous signal, while the on/off state of the device has a Discrete Event mechanism. Methods such as derivative studies are advantageous for locating in advance and with precision these threshold trespassing, enabling for possible performance improvements.
Generally, the strategy for hybrid co-simulations is to conform Continuous Time simulators as Discrete Event ones, or vice versa. At the moment, there is not a standard, in place of

HLA or FMI, specific for hybrid co-simulations.

7.3.2 Best-practice for models integration

Belete *et al.* individuate a structure for the *integration* process.[75] Here, it is intended for co-simulations, but can be extended to a Digital Twin in general, which has to integrate differen hardware and software components.

Pre-integration assessment

An initial analysis on the requirements for integration is needed to sketch a workflow for the implementation. One of the first matters to be considered is the relations between the different components to be integrated, perhaps helping with use-cases and diagrams. The article suggests some other queries to support this preliminary work: to define the objective of the integration, compatibilities of syntax, frequency of data exchange, data format and units of measurement.

Preparation of the models for integration

Models may be implemented on different languages and with different semantics. Therefore, they may require modifications to conform input and output format for the co-simulation. In some cases, it is possible and preferred to adapt the model itself. In other instances, such as commercial programs or big collaborative codes, it is not possible nor recommended, and a custom interface must be developed. This interface may be considered as a *communication wrapper*, intended to adapt inputs and outputs to the common framework of the co-simulation, not much different from concepts seen about virtual sensors.

Modelling the orchestrator

It is important to create the *orchestrator* in this phase. It manages the workflow, from the initialization to the synchronization of the components, as seen before.

Data interoperability

It has to be checked general data interoperability, namely the ability of the components to interpret data coming from different models; in negative case, proper interfaces have to be added, or the existing ones integrated. For instance, the *communication wrapper* can be bi-directional, also become responsible for sending data to the single components.
Factors to pay attention to are interpretability of entity names, unities of measurement and relations between data. A possible tool use is *metadata*, which flags the main data with accessories information. Examples of useful information are: variables meaning, measure units, valid range for the value.
Another expedient is to define and adopt a controlled vocabulary. It narrows the cluster of possible words, in order to give uniformity to the code with a dedicated ontology; the vocabulary may be expanded when needed, with proper precautions such as documentation.

Integration testing

The final step is testing the integration. Generally, a bottom-up approach is used, starting from the components and then adequately testing the final model, but it may also be adopted a top-down strategy. A complex action is to study how the single components uncertainties propagate to the co-simulation and provide a quantification. For open-source academic models, a collaborative community testing may be done exploiting web platforms such as gitHUB.

Interconnection approaches

there are two architectural approaches for interconnections between the components of the co-simulation.
A first possibility is a component-based approach, in which an interface (the orchestrator) accesses all the components, which are independent of each other. The coupling can be implemented as wished, and may be both tight or loose. In case components are situated on different machines, such as remote computing, the coupling is not efficient and can be implemented with different tools.
A second option is a service-oriented approach through web access. This architecture is based on de-localisation and facilitates interoperability, but may lack performance, as well being more complex to code.
Also, hybrid approaches may be adopted.

Performance improvement

For generic co-simulations, the article suggests two hardware methods. The first is to utilize high performance clusters, which allow to run heavy simulations inside institutions. The second approach is to exploit multi-core CPUs with parallel processing; this multi-threading architecture is enabled by specific techniques and may leverage the effectiveness of such simulations.
As long as considering testing and development of the Digital Twin, by employing personal computers, these techniques are helpful. For the final product, a custom-made architecture has to be designed to enable the functioning within adequate performance requirements, with the application of safety margins and needed redundancies.

However, for space applications, hardware has limitations due to high launch costs, reliability and extreme environments protection. This tightens the possibilities for high-performance computation, thus restricting co-simulation complexity and, above all, of heavy models, favouring the use of surrogate models.

7.3.3 Digital Twin integration

Here is briefly illustrated the integration process. Due to the preliminary nature of the project and the limited time, the method has not been faithful to the guide-lines just seen. However, some tips were followed. In future, more attention would be paid to uniformity of variable's naming and measure units, for example.

Software integration

Not many software parts have been integrated. Indeed, the simulated sensors and the surrogate model already were included in the middleware. Often, these two entities are mediated by the *middleware*, that generally has this role (*middle*ware). Here, a simplified architecture confuses *middleware* and *Digital Twin*, while in future updates they could be distinguished.
The only software integration left is the one of the Digital Twin with Arduino. Simulink-Arduino connection was quite spontaneous since it has been part of the design process itself, as described in section 6.3. As extensively explained, the integration was done using *serial communication* as interface: for initialization, it is done by MATLAB; for run preparation and control cycle, by Simulink's package *Instrument Control Toolbox*, with *Serial Send* and *Serial Receive* blocks. The data is exchanged in ASCII format, before being converted into strings and then in numbers.
Many problems emerged coding Simulink-Arduino, but were solved iteratively during the design, leading to the final communication strategy. Instead, during the final integration, the critical point was found in incoherent units of measure, such as sending Arduino *metres* instead of expected *millimetres*; clearly, this was easily solved.
In future developments, sensors will be detached from the middleware, but their integration is already seen in section 7.2.

Hardware integration

Along the lines of what just said, not much integration was done about hardware. Arduino already has a USB port as interface: a USB cable is the only hardware interface. No particular operations were required for this, just connecting it to the PC and checking the port is seen as COM6 (since Simulink needs to work with a specific port).
Generally, hardware integration is more relevant as the project enters its final phases.

7.4 Validation of the model

At some point during the development of the Digital Twin, it is essential to *validate* the simulative model. In subsection 3.1.4, validation is one of the last points. However, it is important that the validation comes along with all the development, in order to correct any configuration errors before it is too late and onerous to change. From this emerges the necessity to adopt ad iterative design method. I

For completeness, *validation* is to check if the designed object, in the broadest sense, is the correct one to respond to the necessities. Therefore, it is needed both a validation process for the model and for the middleware.
For this work, a continuous qualitative validation was performed during the creation of the model, to check if its behaviour was reasonable. Here, instead here is a quantitative validation for the model. Furthermore, the testing campaign in the next sections also assumes the role of validation for the whole Digital Twin (model, firmware and middleware).

Model validation requires the most accurate and numerical approach possible, including iterativity, to bring to a satisfactory model.

As for this work, *validation* process should concern three points.

1. *Numerical tools* validation: in this case, the representation of physical entities and phenomena through equations is done by Adams, as default. Since the software is commercially released, in addition to being used by many users in academia and industry, this point of the validation is not necessary. See section 5.1 for more information about the multi-body model.

2. *Multi-body model* validation: the original multi-body model, described in section 6.3, should be validated, especially in the definition of kinetic and dynamic elements. Qualitatively, it is has been validated. However, for a precise process, it would be necessary to use specific sensors (even external and not integrated inside the Digital Twin) to collect prototype's data to compare with simulations results. Due to time constraints and availability, this part was unfortunately skipped: this represents a critical point in the development.
Moreover, many problems emerged during the surrogate validation, as later discussed. One area for improvement is the definition of motion, which is at constant speed (issue already discussed in subsection 6.3.3). Furthermore, basing motions on constant speed definition speed goes against the representation of contacts in Adams: inaccuracies emerge and the simulations slows down when chaser-target contact happens. Anyway, being this just the first iteration of the Digital Twin project, these improvements are intended for future developments.

3. *Surrogate model* validation: for complete validation, the surrogate model should be validated, being the actual component integrated in the Digital Twin. A formal approach would be to compare it with sensors results, since the surrogate is the actually employed model. However, assuming the multi-body model as validated, it is sufficient to validate the surrogate with its parent multi-body model.
This waterfall approach is the one chosen for this analysis, where validation is limited to checking whether surrogate and multi-body model adhere. However, it is incomplete, since the original multi-body model was not validated, as said.

However, since the multi-body validation misses, a possible and less orthodox strategy is to validate and test the final Digital Twin. If it perfectly works, the surrogate by itself is validated. By testing the overall cyber-physical system, it is not anymore needed to individually test the multi-body model; it would indeed be taken as correct, as the validated surrogate would have been derived from it.
Unfortunately, this was not possible for this work, due to the lack of real sensors for quantitative testing. This can go, again, in a new iteration of the project.

7.4.1 Surrogate model validation

In section 6.3.2, considerations had already been made about the accuracy of the surrogate, for its three polynomials. Two out of three have good values of R^2. Accuracy however only concerns how the polynomials interpolate the set of points used for the fitting. High values of R^2 do not ensure that the fitting is good also for other sets of values (explicitly different from the ones used for building). Having a polynomial, as example, just think to

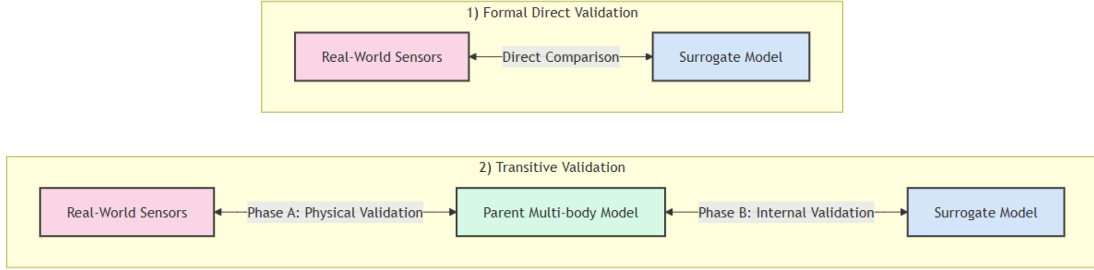

Figure 7.1: Validation approaches for the surrogate model: direct method and transitive method.

Runge phenomenon.
Fromally, considering a surrogate model built with m sets of initial conditions (with $m < M$), *validation* is the process of checking if the surrogate model follows the parent multi-body model, by using the remaining $M - m$ points as test points.

For this work, validation was done by sampling the ranges of validity of the four inputs in 50 points, in order to create 50 sets of initial conditions for an equal number of simulations. The subdivision was done with Latin Hypercube method by Adams Insight, that automatically ran the simulations, as described in section 6.3.3. The process to create the samples is, up to this point, the same used to build the surrogate model, but with a much smaller number of points.
Firstly, a full set of 50 simulations of 0.05 seconds, made with the original multi-body model, was ran. Secondly, the same 50 input sets were submitted to the surrogate model. This was done with MATLAB with a copy of the surrogate model used in the Digital Twin. This gave two full arrays of outputs, one obtained with the multi-body model, one with the surrogate model. By comparing them, it is possible to begin validation process.

Chaser-target distance p_{end}

The first polynomial analysed is the one for chaser-target distance p_{end}. Figure 7.2 compares the results of the surrogate simulations and the original multi-body simulations. If the values lay on the bisector, the two model correspond.
First of all, note that some values exceed 0.1 m, on both the models; the chaser should not be able to go further, since the Linear Guide is just 10 cm long. However, a possible reason is that Adams computes this distance as the distance between the centres of docking: when Δ, linear misalignment, is considerable, this value can be bigger than 0.1 m: chaser-target distance is obtained by Pythagorean theorem as $\sqrt{p_{end}^2 + \Delta^2}$. A more rigorous formulation should be inserted, since this can bring incongruities in the control algorithms of the Digital Twin.
A second clear problem is that the surrogate does not have p_{end} lower than 0.025 m. There is clearly a problem in modelling the situations in which the chaser is close to the target.

The second graph, figure 7.3, compares the outputs given by the surrogate model with the results of the original simulations. The surrogate results are shown with a blue spline.

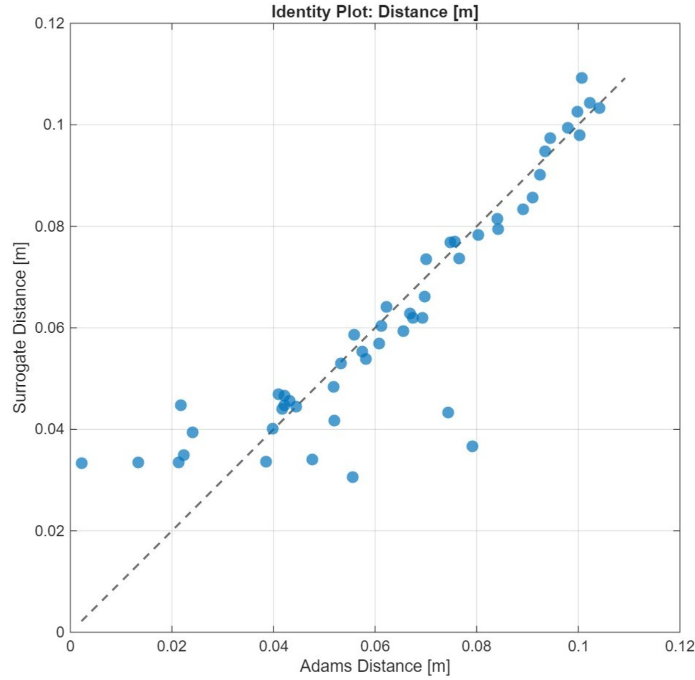

Figure 7.2: Identity plot for p_{end} polynomial.

The spline does not have a particular meaning, since the abscissae show which simulation is considered, not sorted with a particular order; it only has visualization purposes.
Green points are Adams results. Often the polynomial follows the 'trend' of the original model. However, the values are often not correct; this is, as said, especially when the simulation gives $p_{end} < 0.25\,m$.
Moreover, as readable in the title of the figure, p_{end} surrogate polynomial has a NRMSE (Normalized Root Mean Square Error) of 0.1134. This indicator considers the RMS value of the surrogate-parent error, and normalizes it with the distance between the maximum and minimum values from the multi-body model. The formula is:

$$NRMSE = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}}{y_{max} - y_{min}}$$

y_i is the surrogate model output, while $\hat{y}_i$ is the output of the parent multi-body model. The value is rather high, indicating a low quality surrogate. Likely, understanding the reason behind bad performance for small p_{end} could fix many problems of the surrogate.

Finally, figure 7.4 shows the residuals (in *metres*, not normalized). The diagram does not add much informations, since the greater residuals emerge when for missing low p_{end} values. The graph does not reveal particular trends with respect to the inputs, since the run are randomly ordered. Further analyses may reveal anomalous behaviours for specific ranges of the inputs, helping in the refinement of the model. Since this space is only for validation, this investigation is left to future developments.

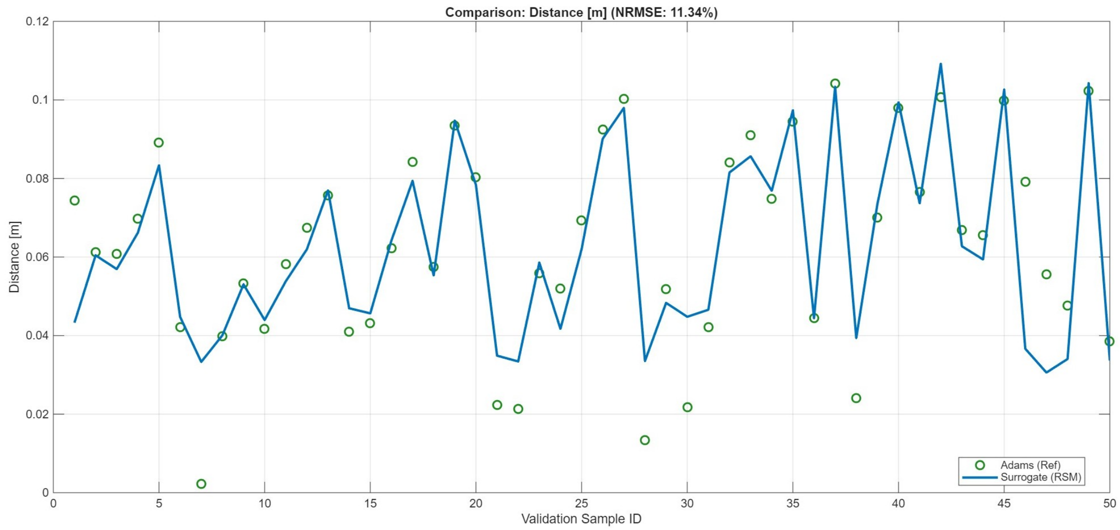

Figure 7.3: Comparison plot for p_{end} polynomial and the original multi-body model.

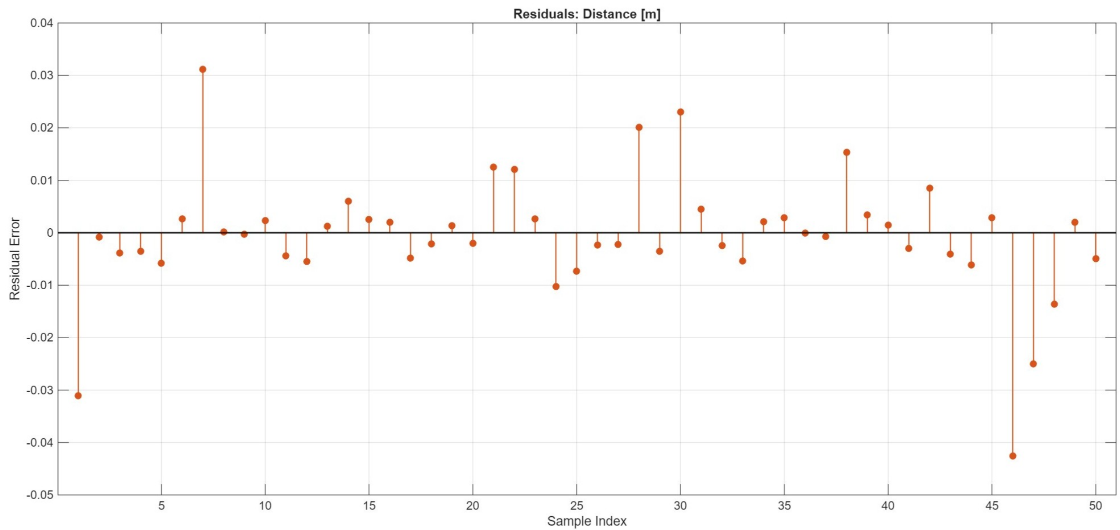

Figure 7.4: Residual plot for p_{end} polynomial.

Chaser speed v_{end}

The second polynomial of the surrogate model is for the chaser speed v_{end}. Again, let's consider the identity plot: on the abscissae there are the values from the actual multi-body simulation; on the ordinates, the estimation from the surrogate.
There clearly is a deviation trend. High and low values of output speed are overestimated, while halfway values are slightly underestimated. However, generally speed is below 0.03 m/s, so the left portion of the diagram is more relevant. Cruise speed, which can be approximately between 1 cm/s and 3 cm/s, is perceived in the trend, but with errors up

to 20%. These errors are quite high, since the chaser moves at constant speed (in the model, not in real cases), so output speed should have the same value of the input: the surrogate model commits relevant inaccuracies in predictions.
Moreover, large mistakes are done for low velocity values, that the surrogate is not able to correctly model, similarly to what happens for small distance values. Probably it means that the surrogate has problems in predicting the contact correctly. A reformulation of the multi-body model and a new surrogate is likely the only way to improve the results. However, by looking at figure 7.9 for $a_{target,RMS}$, these problematic points with poorly modelled low speed values often do not correspond to simulations with a docking contact. See, for example, runs *6* and *14*, which show zero acceleration values in the original Adams model (green points), implying there is no contact with those specific initial conditions and within 0.05 seconds.

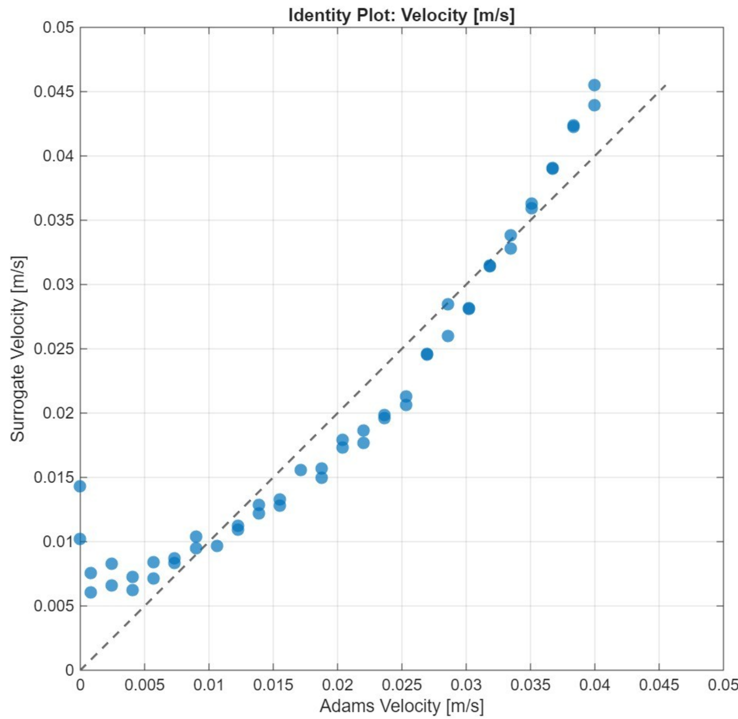

Figure 7.5: Identity plot for v_{end} polynomial.

Passing to the run-by-run comparison plot, more or less the same considerations apply: bad behaviour for low speed values. The blue spline links the results of the surrogate: the choice to use a spline is just for facilitating the comparison with the original model (green points), but does not have a particular meaning, since the simulations (1:50) have no specific order. The surrogate offers a reasonable understanding of the trend velocity has, but not a precise guess of the values, which generally should be constant. Also for speed, an extensive analysis should be done on the variation of inputs.

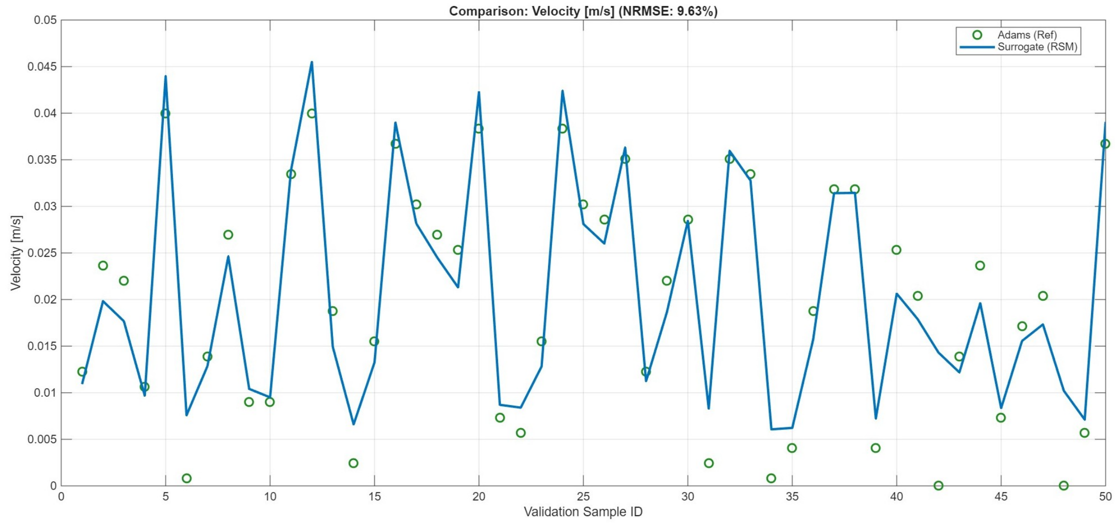

Figure 7.6: Comparison plot for v_{end} polynomial and the original multi-body model.

For v_{end}, NRMSE has a value of 9.63%. Even if slightly better than distance, the value is still high, discouraging the use.

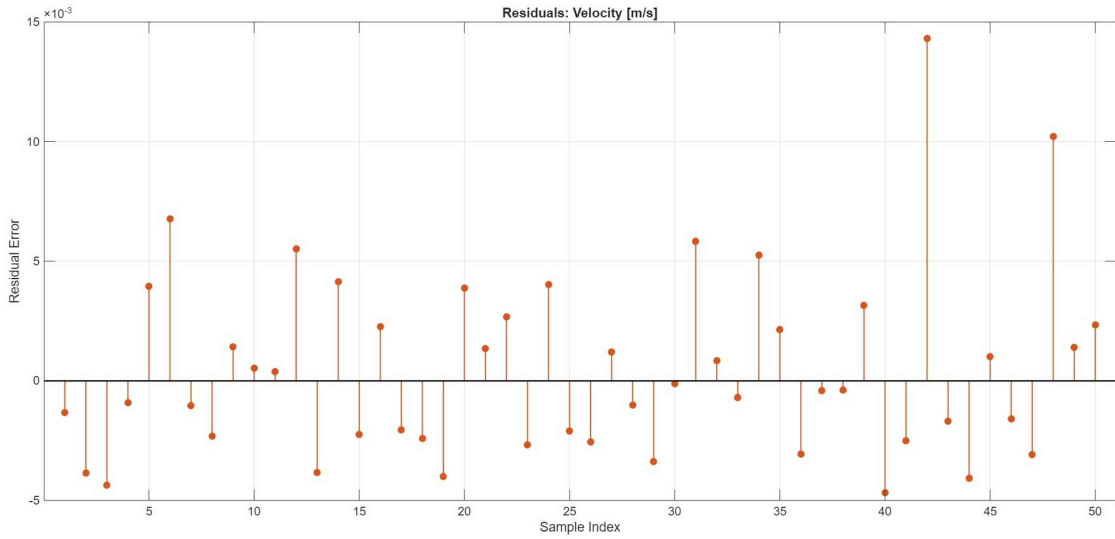

Figure 7.7: Residual plot for v_{end} polynomial.

Target interface acceleration $a_{target,RMS}$

Some considerations may be taken also for the $a_{target,RMS}$ polynomial, which simulates the forces exerted on the interface, in acceleration terms. This polynomial was already excluded for having a mediocre R^2 value.
Analysing the diagrams in figures 7.8 and 7.9, three important mistakes are spotted:

- The identity plot should have the points lying on the bisector for optimal correspondence. The plot does not have any;
- The surrogate is not able to return null accelerations for the simulations without docking impact. In these cases, there is no contact between chaser and target, which does not experience any acceleration: the surrogate is wrong in returning non-zero values;
- The surrogate often outputs negative values for $a_{target,RSM}$: this is a paradox, since a root mean square value is intentionally positive.

Perhaps, the most interesting information to read is the magnitude order of target acceleration during contacts, generally around $0.5\,m/s^2$.

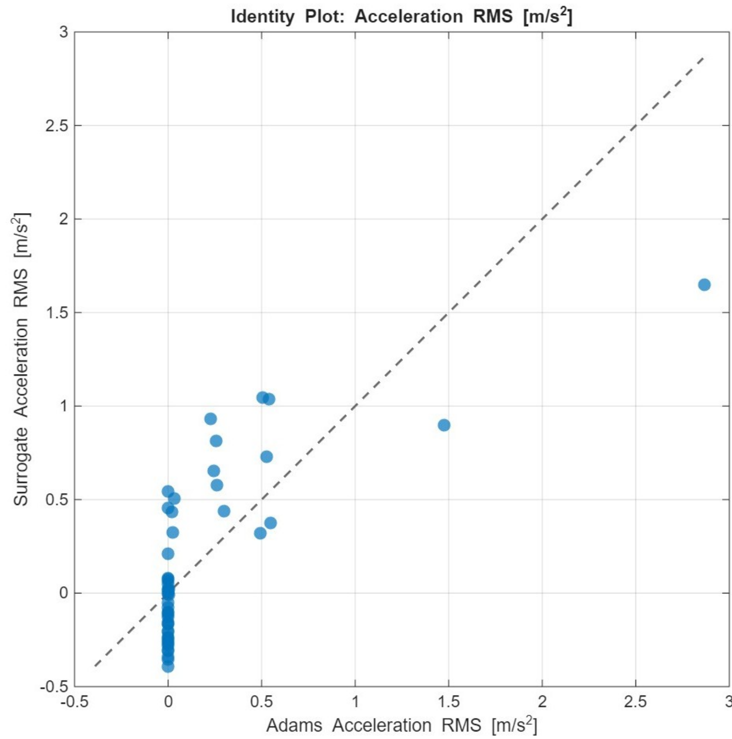

Figure 7.8: Identity plot for $a_{target,RMS}$ polynomial.

For this polynomial, the NRMSE is 11.89 %, not even that much higher than the previous two, demonstrating that R^2 is not reliable for validation considerations. Anyway, in this case both indicate bad prediction capabilities.

Also for $a_{target,RMS}$, the residuals diagram (figure 7.10) does not offer many information; better it would be to replace it with four residuals plots, sorting the simulations as inputs increase (one plot considering p_0 in the x-axis, one considering v_0, and so on for γ and Δ).

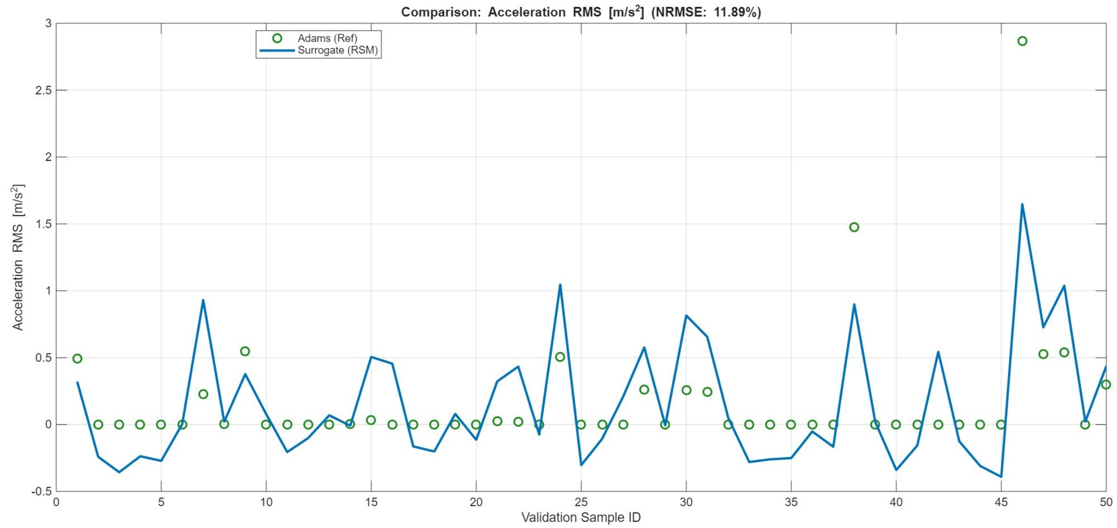

Figure 7.9: Comparison plot for $a_{target,RMS}$ polynomial and the original multi-body model.

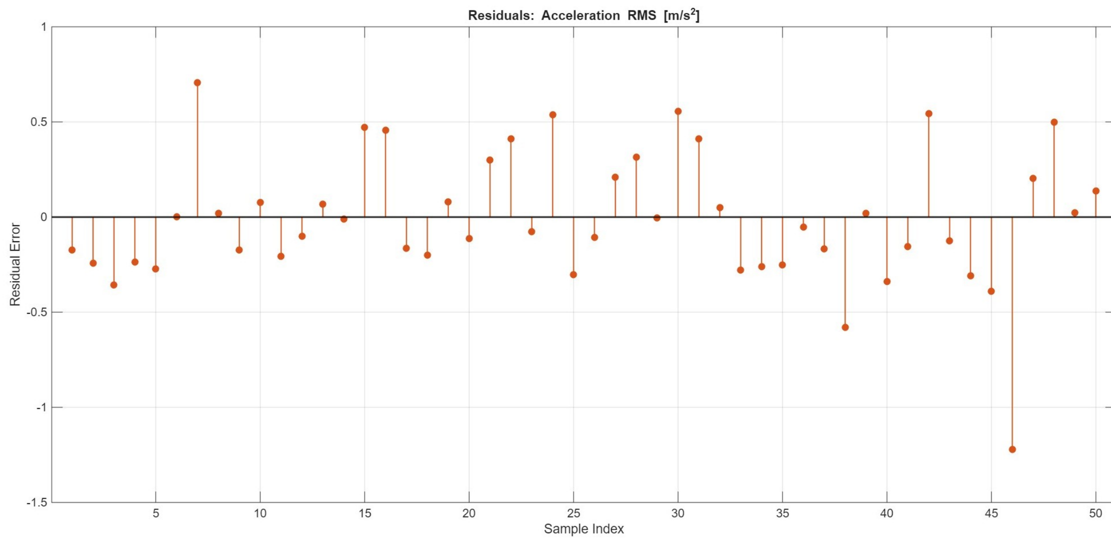

Figure 7.10: Residual plot for $a_{target,RMS}$ polynomial.

Final considerations

To summarise, the main issues are:

- Decent intuition of trends, but bad approximation of the values;
- Predicting capabilities get worse for situations with low values of p_{end} and v_{end};
- The polynomial for v_{end} is slightly better than the other two, as shown already by an higher R^2 value;

- $a_{target,RMS}$ has the worst performances, with important conceptual errors;
- NRMSE values are around 10%, which is not satisfactory. Errors in the order of 10% are rather bigger than the noise of simulates sensors, set up to be around 1%: it is not convenient to use the surrogate model results for filtering purposes.

Clearly, validation is not successful. A better model is surely necessary before an actual use of the Digital Twin.

The refinement should be preceded by a furthered analysis, in particular by considering how the surrogate behaves depending on the inputs. Moreover, it must be reaffirmed that the multi-body model itself has problems, especially when modelling events close to docking contact. Motion is defined as function of speed, which is a problem in launching simulations with contacts, with the solver struggling to perfectly obey to motion laws and contact at the same time. A different definition of motion should be formulated.
Another sign of weakness of the multi-body model is the presence of alarming outliers (whose values deviate significantly from the allowed ranges) in the simulations used to build the surrogate model. See section 6.3.2. Up to this point, outliers have been neglected. Outliers investigation however gives important indications, together with the validation results in this section, before proceeding to model refinement. Also, after the refinement, a preliminary validation of the multi-body model should be done for complete validation, as explained at the beginning of this section.

Other problems of the model are:

- The surrogate model, as it was defined, can only return positive values (magnitude), which is a problem if negative values of speed are returned in magnitude, positive, and should re-enter the surrogate as negative for simulation (as explained in section 6.3.3).
- Low precision of the surrogate is far lower than the one of the sensors. Moreover, since the polynomials are concatenated more times for a one time-step simulation ($T_{step} > 0.05\,s$), these errors amplify, with detrimental effects on accuracy.
- Docking, a non-linear phenomenon, is not correctly modelled in the surrogate model. But also the linear part of the simulations (chaser that moves undisturbed towards the target) is not linear, with speed values changed while theoretically being constant.

At the moment, the analytical model used in the simulated sensor is probably more effective than the surrogate model. Also, considering that, in the multi-body model, speed is an input and is constant for the motion, the output of should be already defined by the inputs. Furthermore, being distances integrable from velocity, the same is for p_{end}. Therefore, the model only introduces errors into otherwise simple predictions. Different is when predicting non trivial quantities as acceleration.
As said, the multi-body model should be refined, and a new surrogate created, possibly with more that 400 training points. Unfortunately, there was no time for this, which is intended for future iteration of the project.

Even if validation is not successful, it was chosen to use the surrogate for a final testing campaign, which clearly has the goal to test the architecture and not the perfect

simulation capabilities of the Digital Twin. At least one iteration is necessary to have a solid result. However, this work contains many prompts for starting with.

7.5 Calibration of the Digital Twin

Calibration is a critical phase. The goal is to tune Digital Twin's parameters in order to make its predictions and commands adhere to the actual behaviour of the system. Each parameter should be calibrated individually, before the final testing. However, in a complex system, parameters are rarely individually independent, and account must be taken of the coupling of the parameters; thus, calibration can be an iterative process up to an optimal configuration.
For this work, the available time was scarce, so only a limited *calibration* process is presented.

Time step calibration

The first parameter to receive attention was the *time-step* length of the real-time control cycle. Initially, the range was restricted to 0.3–0.95 seconds. However, it needed to be better defined and justified.

A short testing campaign was performed to identify the lower bound. The time step duration should be long enough to allow Simulink to perform all the required computations before the next cycle, without taking longer than necessary, with a safe margin. Therefore, its length shall not be too short; otherwise, Simulink would not be able to keep up.

A short test involved running the Digital Twin connected to an Arduino board not connected to the actuators, to test only the software. The cycle's time-step was varied from 0.95 seconds to 0.05 seconds (20 Hz), decreasing by steps of 0.10 seconds. Due to lack of time, a single run for each t_{step} value was performed; precise calibration would require more accurate tests, with multiple iterations.
For the trials, an additional MATLAB function block was introduced in the real-time triggered subsystem (the one in figures 6.12 and 6.16)). Through MATLAB's *tic-toc* function and *persistent* variables, this block monitored the temporal duration of each Simulink's cycle, to see if it was capable of maintaining the desired speed. Saving these durations in MATLAB workspace, it was possible to compare the actual length of the cycles with the theoretical time advancement, as saved in the *log files*. For any test, an average value of the cycle effective duration was computed.
Consider that each simulation stopped when predicting docking contact, with a full duration of 8-12 seconds. Dividing this duration by the length of the desired time-step for the specific run, it can obtained the number of cycles for the full run. More cycles means a longer vector of cycle durations, with more values available to compute the average value for Simulink actual cycle duration. The smaller the imposed t_{step}, the greater the number of cycles and the more precise and reliable the average cycle duration for that run.

The tests showed high variability in the duration of each time-step: cycle duration is actually not stable, with Simulink trying to follow the user's desired value, but with results far from accurate. By way of example, variability in the time-step length can be up to $\pm 10 - 20\%$ of the baseline value, with cycles longer and others shorter.

Simulink tries to slow down the simulation to adapt the pace to the desired one, with the *pulse generator*, that regulates the activation of the real-time cycle *triggered subsystem*, precisely following Simulink's imprecise time counting.
Simulink indeed is a *soft-real* time software, rough in following real-time imposition. For higher precision, *hard-time* tools shall be employed; otherwise, corrections in the simulation and control algorithms should be implemented.

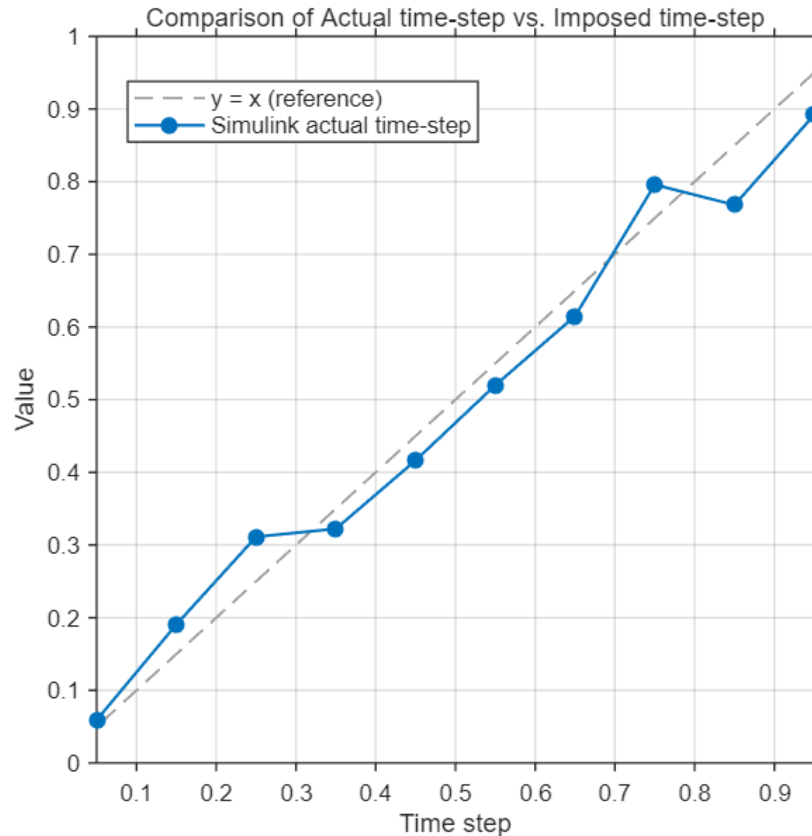

Figure 7.11: Comparison of the theoretical real-time cycle time-step with the actual (average) performed by Simulink.

Figure 7.11 shows how the average cycle duration is followed, varying the desired time-step. The blue dots represent the average actual time-step for the simulation. When over the bisector, Simulink is slower than desired; when beneath the bisector, it is faster than desired. If a blue dot is on the line, real-time is followed correctly.
The diagram has limited accuracy, since each average value was computed, as said, for a single run at a certain time-step value. Moreover, note that all other programmes and files were closed in the computer to leverage performance. Also, initial angular and lateral misalignments were set over the requirements limit, so that the Digital Twin had to compute and send commands also for bring them inside the required range.
The graph shows that Simulink struggles to keep up with the pace for small values of the

interval. However, by decreasing the time-step length, Simulink continues to reduce the actual time-step: it means that a time-cycle of 0.05 seconds still does not represent the minimum value reachable. Indeed, by reducing the time-step, one would arrive at a point in which Simulink would not be able to perform all the computations in the available time, resulting in a horizontal asymptote in the graph. Further investigations are required for this.

Net of a less than meticulous approach, a 0.3 seconds limit was firstly identified as lower limit for the time-step, since for lower values Simulink was not able to keep the time-step under the required value. However, considering the time-step being higher or lower than this value is not the best approach; a better approach would be considering the relative error between the required time-step and the average actual value, choosing a minimal acceptable value for it.
However, a 0.30 seconds value has been multiplied by a 1.2 arbitrary factor, giving 0.36 seconds. Therefore, by excess, the first 0.05-multiple not smaller than this value was 0.40 seconds, which has thus been set as the lower bound for the acceptable time-step range.

On the other hand, the maximum allowed value for t_{step} should be evaluated by a performance analysis. Indeed, increasing t_step, feedback control is less effective: the surrogate model has worse performance, and precision in commands lowers. Therefore, by deciding an acceptable tolerance in the control precision, a limit value could be decided. Actually, the lower t_{step} value is, the better it is. The sense in not using the smallest values in the allowed range is limited. This range is indeed mostly useful for design optimization, rather than for actual use. The only case in which a t_{step} increase can be useful is in case of bad computer's performance (for example with other programs running in parallel) or, in case of spaceflight use, of an energy-saving mode.
Anyway, a superior threshold value for t_{step} was arbitrarily fixed to 1.00 second. Better analyses can be done in future.

Another point that should be investigated is if Simulink and Arduino can proceed in phase, without the emergence of a delay (or an advance). Clearly, this is complicated by Simulink not being able to keep the time step constant. More evolute technologies shall be able to adapt such cycle periods. Anyway, more is said in the next sections.

Filter calibration

There is a second point for which calibration is important, in the current version of the Digital Twin: the α parameter in the filter (see subsection 6.3.3 for details). It weights, in the values used by the Digital Twin for computations, how much the simulated values are considered in filtering the the simulated sensors results. On the contrary, $1 - \alpha$ accounts for the weight given to (simulated) sensor values, in the filtered output.

Originally, a value of $\alpha = 0.3$ was set. As seen, the validation process gave a terrible picture of the surrogate model's performance. Among all its problems, consider that it simulates at constant speed, that cannot correctly simulate when the chaser is close to the target and that its low accuracy is propagated each 0.05 seconds up to the length of the time-step; also other problems emerged.
All these reasons are a clear indicator of how bad it is in simulating the data. The model used for correction is worse than the sensor itself: the mean noise in the simulated sensor

was arbitrarily set as 0.01 of the parameter's maximum value, 1%, smaller than the errors of around 10% in the corrective model). As the filter wants to correct the noise of the (simulated) sensor, the filter is simply not useful. For this reason, lower α values are preferred for the final tests. Clearly, refinements are needed as soon as possible.

In the current work, $\alpha = 0$ perhaps gives better results. However, since the filter is an integral part of the architecture, a non-zero value is maintained, for demonstration reasons. In particular, it was chosen (quite arbitrarily) a value of

$$\alpha = 0.1$$

It means that, in the 'current' values, the simulated sensor accounts for the 90%, while the predictive model for the other 10%.
As actual sensors will be integrated in the prototype and the surrogate model will be refined, an accurate *calibration* process is expected after the updates. It should consider a range of possible α values and the optimal value should be individuated by multiple runs of the Digital Twin.

Other calibrations

Moreover, it should not be excluded that independent optimizations of t_{step} and α give the same result as with a combined optimization: a coupling effect may be present. If there would be time, an optimal combination of (t_{step}, α) could be searched.

Other calibrations that need to be carried out relate to the control laws. Indeed, the *decision maker* module implements chaser's motion control laws that were arbitrarily chosen. These can be adjusted in the formulation and in the parameters, for better functioning. This process is expected in the next iterations of the project, but some considerations are already in 8.2.

Chapter 8

Experimental campaign

8.1 Tests definition

The testing campaign shall be adequately defined before starting it. Often, testing and calibration phases go together, with a certain degree of iterativity. Here, *testing* proposes just as a verification of the architecture. More deepened tests are intended for future phases of the work (as in a V design approach). In particular, for full verification, each requirement (see section 4.5) shall be verified with designated tests.

8.1.1 Basic verification test

The first test is a minimal testing of the overall integration. It purely aims to demonstrate that the architecture is operating and the communication links works. It is just a run, to verify everything works out. For simplicity, input parameters are taken equal to the first run of the testing campaign of the following subsection, as in figure 8.1.

8.1.2 Nominal tests

A testing campaign was planned to test the behaviour of the Digital Twin and the prototype with different initial conditions. Various parameters are varied over a range, to obtain multiple initial configurations, as in figure 8.1:

- t_{span} is the time length of each cycle. Two arbitrary values are taken to note the differences in the behaviour of the prototype: 0.4 and 0.7 seconds.
- $V_{max,user}$ is the cruise speed of the chaser approaching to the target.
- Δ is kept constant and null. The reason is behind a broken wire between the breadboard and the actuator (*Linear Guide 2*): it will be fixed, but there was no time for the planned tests. If the trial is successful for γ, it is taken as effective also for Δ, since the Digital Twin acts at the same way for both the misalignments.
  For future iteration, not only the architecture shall be tested, and Δ shall be included in the campaign. Here, the focus only is on the control strategies.

- γ is the angular misalignment; positive and negative values are taken. With fixed null Δ, positive and negative γ correspond to symmetrical configurations, while varying $\Delta 0$, a coupling effect brings to different geometrical configurations.

These parameters varies and are arranged with many combinations. In the figure, a single try for each configuration is planned, but more for each one are preferred, for robust testing.

NAME	Log file identifier code	Delta (mm)	gamma (deg)	V_max_user (cm/s)	t_span (s)	Result (docking?)	NOTES
T-001		0	0	1,5	0,4		
T-002		0	0	1,5	0,7		
T-003		0	0	3	0,4		
T-004		0	0	3	0,7		
T-005		0	5	1,5	0,4		
T-006		0	5	1,5	0,7		
T-007		0	5	3	0,4		
T-008		0	5	3	0,7		
T-009		0	10	1,5	0,4		
T-010		0	10	1,5	0,7		
T-011		0	10	3	0,4		
T-012		0	10	3	0,7		
T-013		0	-5	1,5	0,4		
T-014		0	-5	1,5	0,7		
T-015		0	-5	3	0,4		
T-016		0	-5	3	0,7		
T-017		0	-10	1,5	0,4		
T-018		0	-10	1,5	0,7		
T-019		0	-10	3	0,4		
T-020		0	-10	3	0,7		

Figure 8.1: Nominal test programme. The schedule includes one test for each configuration, but more repetitions of each test are preferred.

For each trial, many checks can be done. The first is to see whether the prototype completes docking contact in the time predicted by the Digital Twin and with its commands. A visual evaluation of simulated sensor and surrogate model behaviour can be extracted. Other checks can be performed on the speed profile to determine how smooth it is and whether it complies with the user's maximum speed value.

8.1.3 Fault injection tests

A second short test campaign is planned for testing how the Digital Twin responds to a simulated presence of faults. Figure 8.2 shows the description of the trials. Again, a single test for each configuration is planned, but more than one is preferred for solid results. note that Δ is kept constant and zero for the same reasons explained a moment ago.

- *T-101* tests if the DT correctly responds to a broken actuator for misalignment. In fact, MATLAB code shall be temporarily altered to keep constant the angle, without commands for small corrections to γ.

NAME (log file)	Log file identifier code	Delta (mm)	gamma (deg)	V_max_user (cm/s)	t_span	Result (docking?)	NOTES
T-101		0	10 - constant	2	0,4		
T-102		0	0	4	0,4		
T-103		0	0	-3	0,4		
T-104		0	5	Arduino disconnected during the run	0,4		

Figure 8.2: Test description of the trials for evaluating the response of the Digital twin to fault injection.

- *T-102* tests how the Digital Twin and the prototype responds to a high cruise speed. For this test, the tester must bypass the limitations on inserting the desired speed.
- *T-103* tests how the Digital Twin responds to negative speed (chaser moving away from target); again, to inject the fault, speed limitations must be by-passed.
- *T-104* tests how the Digital Twin responds disconnecting Arduino's USB cable.

Each test can offer new ideas for improving Digital Twin protection algorithms. These ideas can be either kept for a new design iteration of the Digital Twin, or immediately implemented: in this second case, the tests can be repeated to evaluate the effectiveness of the corrective actions.

Many other tests can be done, for example by software-simulating faults in other hardware components, such as imposing null prototype's speed to simulate a fault in *Linear Guide 1.*
In successive stages of the project, it can be useful to relate tests to possible real study cases for the docking subsystem. Refer to section 5.1.3 for a first analysis.

8.2 Tests results and analysis

Tests results must be used to validate the Digital Twin or to identify enhancements to the code.

8.2.1 Failed tests

Tests were scheduled as seen, but unfortunately the full campaign could not be completed, mainly due to congenital hardware limitations. As consequence, just simple tests were done to check the correctness of the architecture.

To briefly anticipate the conclusions, the architecture is successful to the extent that basic limitations are disregarded. More details are in the following subsections.
However, big restrictions derive by Arduino's itself. Refining the Digital Twin with successive design iterations is perhaps going towards the wrong direction. There are indeed unavoidable problems, for which the best thing might be to switch to a different hardware approach, replacing Arduino with more professional tools.

8.2.2 Successful basic tests

While the tests listed in the previous section could not be performed, at least the software architecture proved to be effective. Nevertheless, further developments would be required

for completely satisfactory operation, not to mention the various hardware issues.
Figure 8.3 shows the test setup. From left to right, are visible: the computer running the middleware on Simulink; the target (note that the lateral support is stable on one side, and loose on the hidden side), with *Stepper motor* and *Linear Guide 1* beneath (respectively, for γ and motion); the Chaser, at intermediate distance from the target; in the foreground, *Linear Guide 2* (for Δ) is unconnected due to wiring problems.
As said, it was not possible to complete the test campaign. Anyway, some brief tests were performed to test the architecture.

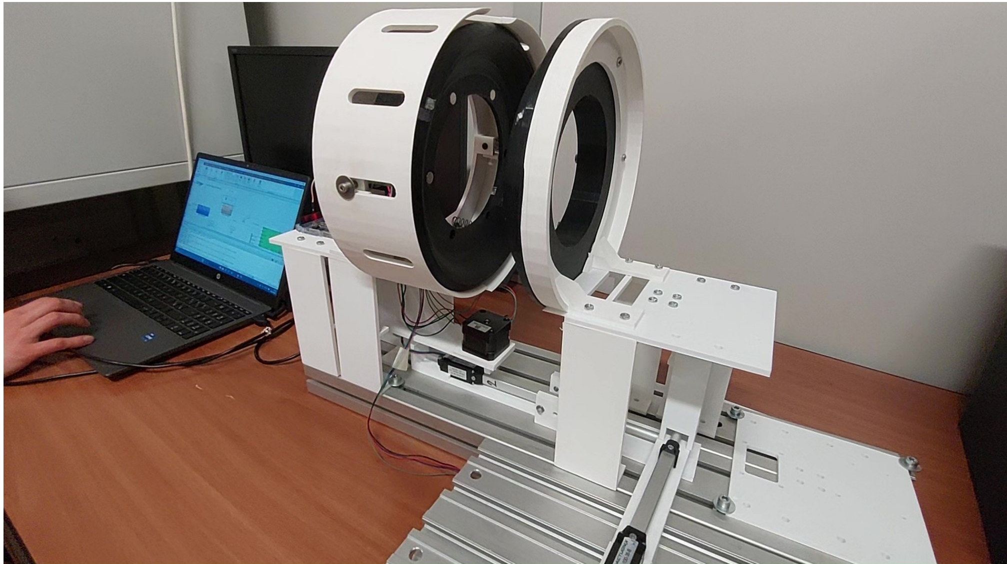

Figure 8.3: Experimental set-up for testing the Digital Twin in the lab.

The first check is for the *connection* between Simulink and Arduino. It has been successful; however, many tests were done along the development of the middleware, leading to many adjustments. Thus, no connection problem was observed.

A second point to check was *homing* process, which aims to bring the actuators in their beginning position and fixing those positions as origin of the reference systems. Homing is done by pushing buttons on the middleware graphical interface, up till the user sees the actuators are approximately in their correct neutral position.
Homing process is quite happening, but with some issues:

- *Linear Guide 1* (for motion): the actuator correctly responds to the commands, which are adequate for docking. However, hardware electric problems affecting the circuits strongly limit the behaviour (see the next section for details). The consequence is that often the actuator extends, correctly going towards the maximum extension, but other times it moves in the wrong direction, going towards the target. This operation is not precise, but the problem is not architectural of the Digital Twin.

- *Stepper motor* for γ: the actuator correctly rotates when asked. However, an architectural issue emerged. The two buttons for homing control are only '*Move*'

and '*Stop*' and this causes stepper motor (which does not suffer of electrical problems) to be homed only counter-clockwise. This makes it impossible to return the chaser to a neutral position when it has positive angles, which would require counter-clockwise (negative) rotations. Another button shall be added for clockwise rotations.

- *Linear Guide 2* (for Δ): the actuator is not wired, as said, thus no test was performed. Being the same model of *Linear guide 1*, it probably would show the same electric problems. Moreover, also for *Linear Guide 2*, a second button for moving in both directions could be necessary, as for the *stepper motor*.

Taking these considerations into account, the architecture is anyway correct, except for the presence of a one-way button for homing of two out of three actuators. Other issues are however described in the following pages.
After homing, it appears that the *Stepper motor* correctly tries to impose the initial misalignment γ_i (the actuation is not effective, due to hardware problems described in the next below).

Finally, real-time phase (the actual control loop from chaser's initial position to docking) was tested, without a rigorous campaign as described in the previous section. Only a configuration was tried, similar to `T-009`.

- γ is correctly adjusted with $\pm 1^o$ commands each control cycle, when the initial value γ_i is outside the requirement values.
- Δ is not adjusted since the actuator is not connected; probably, it would work, exactly as for γ.
- The motion is problematic. The first problem is that, electrically, *Linear Guide 1* almost does not work. Moreover, fundamental problems from Arduino's time management emerged, as explained in the next section. The result is that the first or two cycle see movement, while than the actuator tries to move without results. Moreover, it is likely that the chaser would not make contact within the time predicted by the Digital Twin, which would therefore end the run prematurely.

Here, results from the log file of a simulation (run `Log_2026-03-02_16-04-19.jsonl`) are shown; these results are clearly the commands periodically sent to Arduino and the simulated results, since the actual values do not follow the actuation, as just said.
Initial conditions were:

- $\gamma_i = 10^o$;
- $\Delta = 0^o$;
- $v_{user,max} = 1.5\,cm/s$;
- $t_{step} = 0.5\,s$

Figure 8.4 compares the desired velocity at each time-step (blue step function), which is the goal speed sent each cycle to Arduino, with the speed values as output of the filter (red dots). This second group of points originate from the simulated sensor, thus not

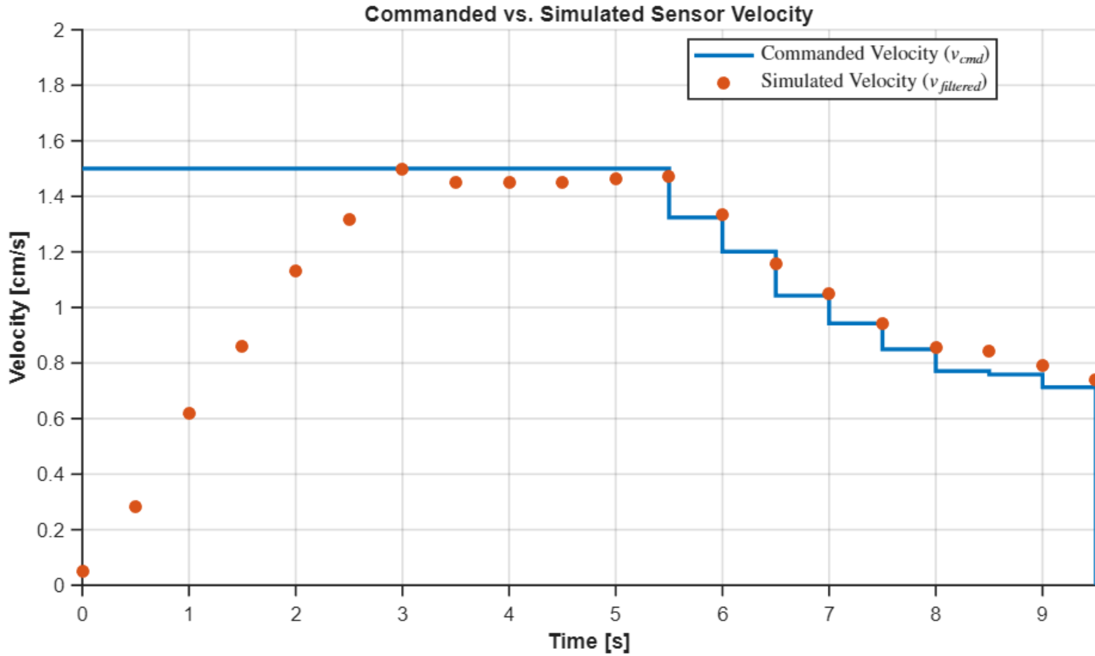

Figure 8.4: Speed values during a simulation are shown. The blue step function represents the objective velocity, that the actuator tries to reach at the and of each step, by translating this value in electrical commands for *Linear Guide 1* (which was actually not able to follow). The red dots represents the output of simulated sensor + filter, at each time instant.

having an actual correlation with reality (which, in addition, is frozen due to not moving hardware). In future, red dots will be replaced by data from actual (filtered) sensors, giving a real feedback on the effectiveness of the Digital Twin.
The red dots show the initial acceleration trend in the first phase of the motion, ignoring sensor uncertainties between 3 to 5.5 seconds during the cruise phase. The maximum reached speed appears to be the user defined one, $1.5\,cm/s$, although unfortunately there is no confirmation from the hardware. The speed is then gradually reduced; the contact would take place at $V_{contact} < 0.8\,cm/s$, which is lower than the $1\,cm/s$ requirement. However, a precise calibration should be done, as mentioned in section 7.5 in order to ensure this requirement is observed also for higher $V_{cruise} = V_{user,max}$: for it, the complete testing campaign would have been useful. Anyway, any consideration should be made after being after ensuring that the prototype correctly implements the commands.
At the end of the simulation, a value of 0 cm/s is sent to Arduino, while the middleware stops running. From this moment on, the idea is that Arduino is still connected to the computer (USB cable), having power to continue its run. Imposing a 0 cm/s speed as objective speed, it would decelerate up to stop. Also this deceleration phase needs calibration in a further iteration of the work, after ensuring how the hardware responds to this stopping strategy.

A second figure, 8.5, shows the evolution of chaser-target distance p, as simulated. In

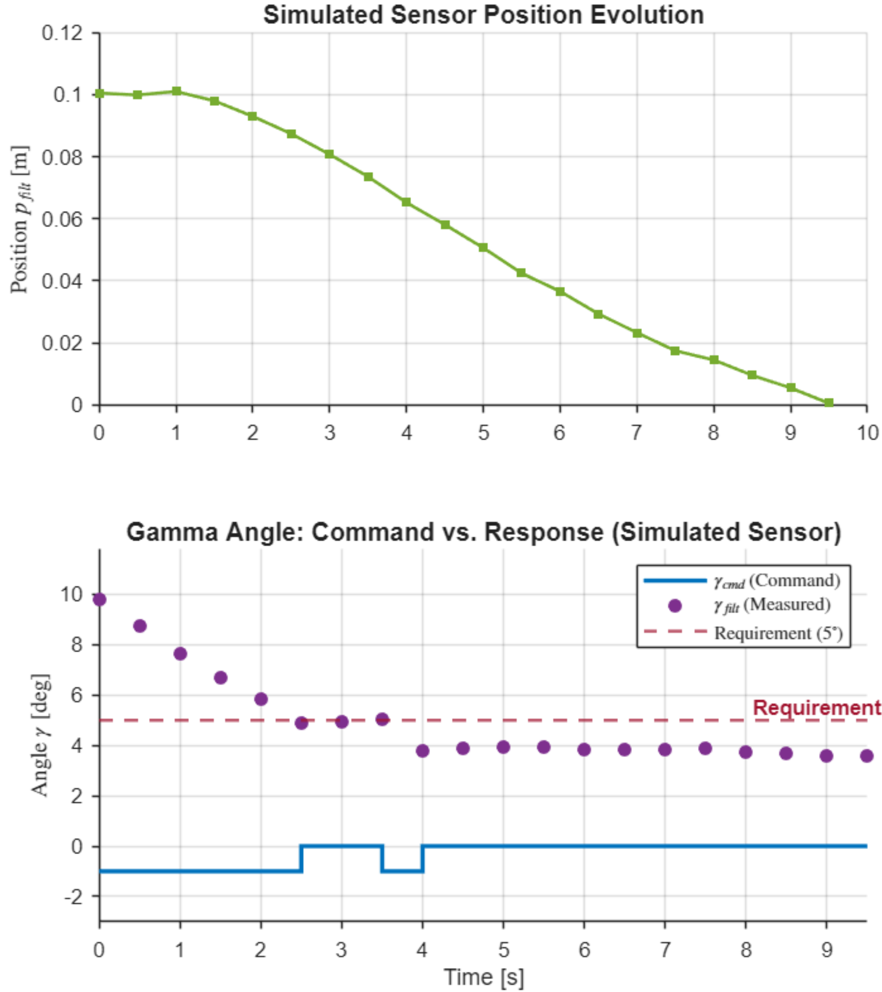

Figure 8.5: Top figure shows the evolution of target-chaser distance during one simulation, with the speed evolution depicted by red points in figure 8.4.
The figure beneath is showing the evolution of angular misalignment γ values, as output of the simulated sensor+ filter; the red dashed line represents the requirement at docking as from Binetti's indications (table 4.4.2). The blue step function represents the commands for the *Stepper motor*.

around 9.5 seconds, the prototype should dock. However, due to actuator faults, there was no possibility to receive a visual feedback from the prototype.
The bottom image, on the other hand, represents the evolution of γ, along the lines of what seen for speed. The difference is that, for the misalignments, the actuators impose 'instantaneous' small movements, of $\pm 1^o$ for γ, at the beginning of each time-step, as shown by the blue line. These small movements are, for γ, correctly implemented by the actuators, as verified by eye.
Also, γ simulated value suggests a correct functioning, even if complete tests on the hardware should verify if the implementation exactly corresponds to the theory.

However, a correction on the command strategy could be made. When, at 3.5 seconds, the sensor has a positive fluctuation due to artificial noise in the simulated sensor, the Digital Twin sends a command to reduce the angle. The overtaking of γ limit value is not real, with the final angle being 4° instead of 5°. A more precise strategy could be implemented to manage the commands for misalignments. Anyway, reaching chaser-target contact with an angular misalignment lower than the required values seems to be ensured, even if a confirmation in the real prototype is required in future.

The Digital Twin seems to correctly implement commands to dock within the specified constraints; however, complete testing varying the initial conditions is required, This will probably lead to a refinement in the control strategy. Nevertheless, there was no feedback from the actual prototype, not correctly functioning. Even after fixing hardware problems, predicted and actual behaviour may not necessarily coincide.

8.2.3 Limitations: why the Digital Twin is not working

Let's start from the fundamental conclusion: the architecture works, the Digital Twin doesn't. Therefore, although the work of building a solid framework was correctly performed, hardware problems and one congenital issue are limiting the Digital Twin, which did not work. Time available has not been sufficient to overcome these limitations.

Electronic problems

The first class of problems emerged from the moment the prototype was reactivated after Binetti's work ended. Many wiring connections were fixed and batteries changed with a bigger set (eight 1.5V AAA batteries instead of six).

However, a problem was intensely investigated but not solved. Arduino receives commands from the computer and sends them to a secondary circuit, which receives Arduino's digital commands and power supply and converts it in precise commands for the actuators. This circuit is based three *A4988 drivers*, responsible for this processing of voltages. The output signals for the two linear actuators are not adequate for a precise control: they move, but only sometimes, often just for few instants. At other times, the direction of motion is wrong, while often the motor vibrates and stays still. however, this kind of problems did not affect the *Stepper Motor* for γ.
Many tests and regulations were tried, on the power voltage from the set of batteries and on the reference control voltage of the drivers (V_{ref}). Nevertheless, a solution was not found. A possible reason is an alteration due to static currents, but an answer was not found.
A practical solution would be to replace the drivers, but there has been no time for it. If the work were to continue, this should be the first step.

However, it is also possible that Arduino's firmware is not correctly calibrated in translating speed command into Arduino's digital outputs. For this, in a future development, the firmware should be checked and adapted for a better translation of commands into electric signals for the motor drivers.

Moreover, a jst pin connector was missing, this reason as behind the Digital Twin tested without actuator 2 (*Linear Guide 2*) active for Δ control.

Mechanical problems

Two main mechanical issues emerged during the development of the DT and the testing. Both combines with electrical problems, but shall receive a revision of the prototype's design.

The first issue is that a transmission screw does not mesh with the actuator shaft (*Stepper motor*). The consequence is that the stepper motor is not able to transmit rotations to the chaser, without an actual implementation of γ variations.
If this screw is hold tightly against the shaft, the latter does not move, unable to win the starting torque. On the contrary, if the screw is loosened, the shaft rotates freely: it is possible to appreciate the effectiveness of Digital Twin's misalignment control, but it is not actually applied to the prototype.
This transmission shall be designed from scratch, since it must be able to rotate all the block composed of the two *Linear Guides*, their supports and the whole chaser.

A similar problem derives from *Linear Guide 1*, responsible of motion. After fixing electric problems, perhaps some mechanical adjustments shall be made. The frictions between chaser and the metallic floor are significant, so the actuator could benefit from their reduction.

Arduino and Digital Twins: the primal problem

The main weakness of the Digital Twin is here. The motion, from what little we saw in the tests, is jerky. The reason is simple, and cannot be completely fixed. During its loop, Arduino has to understand which part of the *loop* is logically active, then check for commands update from Simulink, translate them into decisions, possibly moving *Linear Guide 2* and the *Stepper motor* for misalignments adjustment, and only at that point moving the chaser. But Arduino is single-core: it can do only one action at a time.
This is not a problem for discrete movements as those for misalignment control. It is instead not adequate for controlling the speed, which should be continue for all the approaching motion: at each iteration of the loop, the chaser stops for a fraction of second, necessary for all the other actions listed above; then, the actuator must win again the starting torque and restart the motion.
As already seen in section 6.3.4, changes can be made, in order to obtain better motion for the chaser. One possible strategy is even using sub-loops (with, for example, *while* cycles inside the *void loop*): this configuration would see Arduino starting a unique long cycle, containing all the instructions in its first iteration; at operations completed, the *loop* iterates infinitely, but without performing actions. This could be done by using, for instance, a boolean variable *simulation_done*, false for the first iteration of the loop, during which activate all the instructions, and true for any successive iteration (activating the empty portion of the *loop*).

This limitation is congenital of Arduino and cannot be changed. If anything, it is possible to simplify Arduino's firmware, in order to reduce the duty cycle for which the chaser is not moving during each control-cycle. Other strategies are to use multiple Arduino devices, one for each motor, complicating the communication with Simulink. Possibly, the USB cable should be split in three to communicate with three Arduino

modules, but this is even more complex and perhaps not possible.

So, it is not possible to completely overcome this huge limitation. Two paths are possible for future developments.
The first is to refine the current design, by fixing the hardware problems and submitting the Digital Twin to a second iteration, since the current architecture shows to be solid. Nevertheless, the problem of non-continuous motion is not avoidable. Also, the surrogate model and the control algorithm shall consider that for a fraction of the time-step, the chaser is not moving, otherwise not correctly modelling the real behaviour.
The second path, suggested in case of a development of the overall docking project, sees the replacement of Arduino with more powerful hardware. However, this profound change needs a reformulation of firmware and middleware, perhaps with Simulink not suitable. After these changes, it would be possible to proceed to another iteration of the design, with the goal of a usable Digital Twin.

For this work, Arduino was used anyway for two reasons. The first is that the prototype relied on Arduino from Binetti's work and there was no time for a change. The second reason is that there was a lack of experience on the subject, and certain limitations were not clear at the beginning. Furthermore, it was not possible to verify how much these limitations affected the behaviour of the motor.

8.2.4 Requirements verification

In this last summary, Digital Twin requirements, defined in section 4.5, are recalled. The objective is to review them retrospectively, to identify which ones have been met and which ones have not. The verifications are not formal: at advanced design iterations, the testing campaign should have the goal of verifying each requirement.

- *R-INTF-1*: the DT correctly collects data from the simulated sensor; successive iterations shall see the integration of real sensors.
- *R-INTF-2*: the DT correctly collects data from the simulator, which is the surrogate model, fully integrated in the Digital Twin.
- *R-INTF-3*: the DT correctly logs sensors data and state information at each cycle.
- *R-INTF-4*: the DT correctly logs simulation data at each cycle (currently only p value).
- *R-INTF-5*: this requirement is not met; only the last step is available to the Digital Twin, for feedback purposes. Future developments can work towards accessibility to older data, which currently are only stored.
- *R-INTF-6*: the DT sends commands to the prototype correctly; however, mainly for hardware issues, the commands do not bring the desired result.
- *R-INTF-7*: the middleware sends data to the simulator (the surrogate model) each cycle.

- *R-INTF-8*: an user can run the Digital Twin and move the prototype by means of it. However, hardware issues represent strong limitations.
- *R-INTF-9*: an user can fully access the state of the physical twin a posteriori. Work can be done to allow data access in real-time.
- *R-PERF-1*: the DT is able to guarantee cycles smaller than 1 second; currently, up to 0.4 seconds it is considered effective, but with fluctuations, t_{step} can be brought up to 0.05 seconds.
- *R-PERF-2*: the DT seems to satisfy prototype's requirements, as seen in this section; however, a more detailed analysis is expected to consider a broader range of initial conditions. Moreover, the control strategy could not be verified on the hardware.
- *R-PERF-3*: this requirement could not be verified, since hardware problems did not allow to complete docking. However, as Arduino is not fully adequate for a DT, this requirements is at risk.
- *R-FUNC-1*: new sensors can be easily added by changing the Digital Twin with slight modifications. However, an autonomous modulation in the type and number of sensors is not possible.
- *R-FUNC-2*: even if still simple, a first health status check is performed each cycle.
- *R-FUNC-3*: currently, this was not possible to be verified. A slight hardware modification shall require the surrogate model to change; nevertheless, these variations can be made without dramatically changing the DT, at least in its architecture.
- *R-FUNC-4*: the DT simply is able to stop the simulation if it detects a possible fault.

Chapter 9

Integration with the LPR

This work focuses on Digital Twins for space applications, but up to now all was about general methodologies and a test on a simple laboratory prototype. Passing from this to a space oriented product requires, first of all, a full team working on it. However, the design process, although much more complex, can start from the same method followed in this work.

Before moving into real space applications, such as the *Lunar Pressurized Rover* (section 4.2), the first necessity is to move towards operational docking solutions. Ideas can come from section 3.3.

Personal considerations and observations emerged during the work and are here listed:

- Suspensions and perhaps active hydraulic or electrical actuators are necessary for compensating terrain undulations and weight distribution inside the rover, adding active pitch control.
- The area in front of the docking passage of the lunar habitat can be paved or flattened for smooth terrain, but maintenance would be needed.
- Emergency situations shall be considered, at least in the global architecture of the rover, such as with the possibility to exit the rover with EVA suits in case of faults. Redundancy and maintenance capability are key factors in the design of a manned vehicle.
- The most likely approach for terrain docking is the use of single or double Steward-platforms, improving the solutions already existing for free-space docking solutions.
- Docking has always been performed by frontal approach, but tangential solutions could be explored. A creative example could be a docking port on the ceiling of the rover, linking onto a raised floor of the lunar habitat, with a stair for walking (as entering a submarine).
- One point to be considered in the design phase is the protection from *lunar dust*, which can jeopardise material duration and watertight seal.
- Should not be forgotten neither the *undocking* operation, which could be active or mostly passive.

Probably, the major area of interest for Digital Twins is the rover's approaching strategy, for which there is the greatest margin for optimisation. Indeed, misalignment reduction is perhaps the most important aspect for the success of docking, as well for offering the docking interface easy alignment configurations that can reduce the stress on the components.
The research of the approach strategy combines optimal path algorithms, guidance, navigation and a perfect control of the locomotion system. For this, machine learning can offer interesting solutions, as well as learning models for preventive simulations.

A second consistent difference between a table prototype and an actual docking platform for lunar application stands in the design constraints.
First of all, the list of materials to draw from is very limited, since protection from extreme environments is required. Moreover, structures shall resist to high stresses also during non-operative phases such as launch and lunar landing. Another complex design area is for the electronic components, which shall resist to a wide spectrum of radiation and must be adequately cooled.
Secondly, there are many budget constraints. Launch costs, which are still exorbitant for passing L1 point and reaching Moon. Energy and power budgets for electronic subsystems, with the Digital Twin being one of these. And again, the thermal budget, since each electronic component shall be cooled.

To begin the development of the final version of the Digital Twin for the docking system, nor Arduino nor Simulink are appropriate: a custom approach is probably the way. However, development method and the architecture may be the same used in this work, but considering a significantly large campaign of *Validation, Verification & Testing* (VV&T).
Another point to keep in mind is that the Digital Twin will probably be included in a global Digital Twin of many subsystems of the rover. From an hardware perspective, it will probably be included in the main avionic subsystem.
One feature common with this project is the on/off functioning: often the docking Digital Twin could be in a stand-by mode, since not necessary, supporting the systems budget management.

Chapter 10

Conclusions and further developments

The current work tried to achieve three goals. Firstly, to clarify the concept of Digital Twin, tracing a brief review of the state-of-the-art, with a specific focus on space exploration applications. This first portion of the work delineated a technology still in its early phases, often misinterpreted, but with great potential, especially in giving autonomy to future missions.

Consequently, the second purpose was to trace a methodology for developing a Digital Twin, in order to pave the way for future projects. A procedure was defined, consistent with the iterative design approach behind many engineering projects, that starts from the preliminary definition of the problem and proceeds to development, integration and a VV&T campaign (Verification, Validation and Testing). At any point, it may emerge the urge to come back to previous steps, until the Digital Twin is ready for operations.

The third objective was to demonstrate the validity of this method through the realization an actual Digital Twin. In particular, it was chosen to integrate this strategy with the overall project in which the thesis collocates: the *Lunar Pressurized Rover*. Thus, it was realized a first version of a Digital Twin for an already realized prototype, built for testing novel technologies for the docking of the wheeled pressurized vehicle with a stable Lunar habitat.

The developed software only represents, theoretically, the first iteration of such a project. Indeed, it partially lacks in good functioning, net of hardware failures that could not be fixed within the time available. One of the biggest limitations of the Digital Twin, however, is not the design methodology, but rather the choice of Arduino as electric controller for the the actuators, as the prototype had been designed. Arduino indeed cannot perform more than one action simultaneously, due its single-core nature, limiting the smoothness of motion control. Refinement to this version could be done, as later discussed, but Arduino should be replaced for an operative Digital Twin.

Despite the non satisfactory results in terms of usability, the project is successful in proving the effectiveness of the proposed design method. For more clarity, each section of this work starts with additional methodological indications for the reader, focusing on that specific step within the overall process.

The first draft of the Digital Twin has indeed been created in few months, following the guidelines defined in the first part of the work. It is based on a Simulink model, acting as core, both with decisional capabilities, both as middleware and as human interface. The model used for simulating the system is a surrogate model built over a multi-body Adams model; the surrogate model, however, has shown strong inaccuracies, and should be refined for future use. In addition, instead of installing actual sensors to ensure feedback from the system, simulated sensors were built, made of a simple algebraic model with noise addition.

The software architecture is fully functional, demonstrating the validity of the proposed and followed design approach, as proven by the successful verification of a good portion of the requirements. This should be considered the primary result of this thesis.
On the other hand, many criticalities were encountered, that prevented the Digital Twin from being considered ready for operation. Hardware faults, Arduino's limitations, surrogate model inaccuracy and the adoption of simulated sensors are the main actors in the deficient behaviour. Since the Digital Twin development could receive limited effort, there is space for many improvements. The lasted around six months, but only three saw the actual development of the Digital Twin, with just one month and half actually working on Simulink's middleware.
The thesis contains many hints for a possible refinement, specified in all the sections of active development. A summary is now given, serving as a starting point for a subsequent iteration.

10.1 Following developments

The Digital Twin's inadequate performance does not affect the validity of the work, for which full operation of the Digital Twin could be considered a secondary goal.
Future work could start from the current version of the Digital Twin. It indeed could not be used to perform a full testing campaign, conceived to test its behaviour in response of varying parameters, such as the approaching speed of the chaser docking interface towards the target one, or the initial geometrical misalignments.
At least one more design iteration is necessary to have a fully functional product for the prototype; furthermore, significant hardware issues have been encountered, requiring careful hardware modifications. At that point, the resulting Digital Twin could be used as starting point for adapting to successive evolutions of the physical prototype, with additional iterations.

These considerations should not scare, since iterativity is a necessary feature in the design process, also for Digital Twins. Important section 3.1.4 highlighted this from the beginning. As hints for an improvement are included in many sections of this work, a brief summary of the most relevant is:

- *Arduino*, which represents the main limitation. It is unable to perform more than one action simultaneously, being single-core. This point can be ignored while working on a simple prototype, but must be addressed moving on more concrete projects. Note that replacing Arduino with other solutions may force to redesign considerable portions of the Digital Twin.

- *Arduino's firmware* is the most neglected part of this work: an optimization could give substantial results in smoothing the control cycle of the prototype.
- *Simulated sensors* shall be replaced with actual *sensors*, for actual feedback; this is necessary for a fully operational DT, otherwise a qualitative validation must rely solely on visual inspection.
- The *Multi-body model* and its *surrogate* showed poor simulating capabilities: a revision is essential, as explained during validation (section 7.4).

Anyway, the global design procedure is effective for the development of any Digital Twin, even the more complex ones. The project was only a first draft: a final compliance analysis of the requirements in section 8.2.4.

In addition, extra functionalities can be added from scratch, adding higher levels of the pyramid in figure 4.8. Some functionalities are here suggested:

- It would be useful to enhance the *portability* of the Digital Twin, in order to easily export and use it other computers.
- After perfecting the Adams model, a useful function could be to launch, at the end of each Digital Twin run, a full simulation of the complete multi-body model to have a precise estimation of the dynamic actions during the docking contact, given the kinematic conditions just before the impact.
- Analogously, acceleration sensors can be added to measure the force acting on the target during the impact, as quickly analysed in section 6.2.
- An important function would be the capability of the Digital Twin to read the log files to access previous data, as also sketched by the requirements (*R-INTF-5*). Also sensors filtering could benefit from this.
- It could be envisaged the possibility for the user to access real-time data during the motion, with information displayed, for example, in Simulink's Diagnostic Window. This point is however secondary and may slow down the middleware: pros and cons shall be evaluated.
- Advanced algorithms can be implemented, as in section 6.6. A particular focus should be onto optimization and fault prevention.
- On the same line, an 'intelligent' model could be created, providing the surrogate model of the capability to learn from past experience. The training could be done real-time or, more easily, off-line by using data from ended old trials. A similar approach could be implemented for precise filtering.

Chapter 11

Appendix A: Arduino firmware code

```
// ATTENZIONE !
// Identificativo usato da Simulink/MATLAB per capire se il firmware
    caricato su Arduino sia quello giusto
const String FIRMWARE_ID = "VER_1_1_1";


// GENERAL INITIALIZATION OF TIME-STEP; PARAMETER UPDATED BY SIMULINK
float dt = 0.4; // 300ms in secondi.


// Valori iniziali di gamma e delta; mandati da MATLAB e aggiornati
float initial_gamma_ang = 0.0;
float initial_delta_mm = 0.0;

// Valore iniziale della velocit
float speed_max_user_input_mms = 0.0;

bool parameters_confirmed = false; // Stato iniziale di inizializzazione
bool homed = false;
bool config_applied = false;
bool real_time_started= false;

int motor_index = 1;        // Parte dal motore 1. Serve per far fare
    homing a tutti i 3 motori.
float current_pos = 0.0;    // Posizione temporanea per l'homing

float delta_current = 0; // mm
float gamma_current = 0; // degrees


// --- VARIABILI PER IL REAL-TIME (PUNTO D) ---
float target_speed_mms = 0.0;      // Velocit  desiderata da Simulink
float current_speed_mms = 0.0;     // Velocit  attuale (rampa interna)
float acceleration_mms2 = 5.0;     // Accelerazione "dolce" (mm/s^2)

```

```
34 unsigned long last_step_time_motor1 = 0;   // Timer per i passi del Motore
       1
35 unsigned long last_ramp_calc = 0;          // Timer per il calcolo della
       rampa
36 unsigned long last_serial_read = 0;        // Timer per non intasare la
       seriale (opzionale)
37
38 // Variabili per tracciare i target ricevuti
39 float target_delta = 0.0;
40 float target_gamma = 0.0;
41
42
43 float current_main_pos_mm = 100.0; // Inizializza con la posizione di
       partenza (es. 100mm)
44 float active_accel = 0.0;          // Verr  calcolata ad ogni nuovo
       comando Simulink
45
46
47
48
49
50 // ------------------------------
51 // PIN DEFINITION
52 // ------------------------------
53 const int STEP_PINs[] = {0,9,6,5}; // PWM PINS (TBChanged)
54  // Questi pin sono utilizzati per inviare segnali di impulso al driver
       del motore.
55 // I valori 0, 9, 6 e 5 rappresentano i pin digitali dell'Arduino a cui
       sono collegati i pin STEP.
56 const int DIR_PINs[]  = {0,8,7,4};
57 // Questi pin controllano la direzione di rotazione del motore.
58 // I valori 0, 8, 7 e 4 rappresentano i pin digitali dell'Arduino a cui
       sono collegati i pin DIR.
59
60
61 // Microstepping PINs
62 bool microstepping_enabled = false;
63
64 // const int msPIN_1 = 11;
65 // const int msPIN_2 = 12;
66 // const int msPIN_3 = 13;
67
68 const int EN_PINs[] = {0,11,12,13};
69
70 // ------------------------------
71 // ERROR INTRODUCTION
72 // ------------------------------
73 // Definizione di un array di float per le variazioni di posizione (deltas
       ).
74 // Questi valori rappresentano le distanze in millimetri (mm) che l'
       attuatore deve percorrere.
75 // In questo caso, le variazioni sono 0 mm e 10 mm.
76 float deltas[] = {0,10}; // mm
77 // Definizione di un array di float per le variazioni angolari (gammas).
```

```
78 // Questi valori rappresentano gli angoli in gradi (deg) che l'attuatore
     deve raggiungere.
79 // In questo caso, le variazioni sono 0 gradi e 5 gradi.
80 float gammas[] = {0,5}; // deg
81
82 bool DEBUG_MODE = false;
83 // Variabile booleana per attivare o disattivare la modalit  di debug.
84 // Se impostata su true, il codice potrebbe stampare informazioni di debug
      utili per il monitoraggio.
85 // Se impostata su false, le informazioni di debug non verranno stampate.
86
87
88 // ------------------------------
89 // MOTION PARAMETERS
90 // ------------------------------
91 // Definizione della velocit  nominale in millimetri al secondo (mm/s).
92 // Questa    la velocit  a cui l'attuatore dovrebbe muoversi durante il
     funzionamento.
93 const float v_nom_mm_s     = 10.0; // mm / s      : Nominal velocity
94 // Definizione del passo della vite in millimetri per rivoluzione (mm/rev)
     .
95 // Questo valore rappresenta la distanza percorsa dall'attuatore per ogni
     giro completo della vite.
96 const float screw_pitch_mm = 2.0;  // mm / rev    : Pitch
97
98 // Definizione del numero di passi per rivoluzione del motore passo-passo.
99 // Questo valore    importante per calcolare il movimento dell'attuatore
     in base ai passi del motore.
100 int steps_per_rev          = 200;  // steps / rev   (TBC)
101
102
103 // Timing for each phase
104 // Definizione dei tempi per ciascuna fase del movimento dell'attuatore.
105 // Dt_I: Tempo di accelerazione in secondi.
106 const float Dt_I   = 0.5;    // Ramp up time in seconds
107 // Dt_II: Tempo di velocit  costante in secondi.
108 const float Dt_II  = 11.0;   // Constant speed time in seconds
109 // Dt_III: Tempo di decelerazione in secondi, uguale al tempo di
     accelerazione.
110 const float Dt_III = Dt_I; // Deceleration time in seconds
111
112 // Other time definition
113 int undocking_time = 5; // s // Tempo di "undocking" in secondi,
     utilizzato per pause o operazioni specifiche.
114 int wait_time      = 5; // s // Tempo di attesa in secondi tra le
     operazioni.
115
116
117 // ------------------------------
118 // SPECIFIC ACTUATOR PARAMETERS
119 // ------------------------------
120 // Linear Guide 1
121
122 // Linear Guide 2
123 // Current Position (y_d)
```

```
// Posizione attuale dell'attuatore in millimetri (mm).
// Questa variabile tiene traccia della posizione corrente dell'attuatore.
float current_position_mm = 0.0; // mm

// Stepper Motor
// Current Angle (phi_d)
// Angolo attuale del motore passo-passo in gradi (deg).
// Questa variabile tiene traccia dell'angolo corrente del motore.
float current_angle_deg = 0.0; // deg


// -----------------------------
// FUNCTIONS
// -----------------------------
// stepMotor(STEP_PIN) : moves the stepper motor connected to the driver connected to the STEP PIN
// muove il motore passo-passo collegato al driver
// attraverso il pin STEP specificato dall'indice 'motor'.
// Questa funzione invia un impulso al pin STEP per attivare il movimento del motore.
void stepMotor(int motor)
{
  // Imposta il pin STEP del motore specificato su HIGH per generare un impulso.
  digitalWrite(STEP_PINs[motor], HIGH);

  // Attende per un breve periodo (100 microsecondi) per garantire la larghezza minima dell'impulso.
  delayMicroseconds(100);          // Minimum pulse width

  // Imposta il pin STEP su LOW per terminare l'impulso.
  digitalWrite(STEP_PINs[motor], LOW);

  // Stampa un messaggio di debug sulla seriale per indicare quale motore   stato attivato.
  // Serial.println("!!! DEBUG : Moving stepper connected to PIN: " + String(STEP_PINs[motor]));
  }


// -----------------------------
// chaser_move(): Moves the chaser through the target
// La funzione accetta un parametro booleano 'forward' per determinare la direzione
// e un parametro 'motor' per specificare quale motore utilizzare (default 1).
void chaser_move(bool forward = true, int motor = 1)
{
  // Set direction
  // Se 'forward'   true, il pin DIR sar  impostato su HIGH; altrimenti, su LOW.
  digitalWrite(DIR_PINs[motor], forward ? HIGH : LOW);

  // Enable Motor
  // Abilita il motore impostando il pin ENABLE su LOW.
```

```
digitalWrite(EN_PINs[motor], LOW);

// Compute nominal steps per second
// 'steps_per_mm' rappresenta il numero di passi per millimetro.
// 'v_nom_steps_s' rappresenta la velocit  nominale in passi al secondo.
float steps_per_mm = steps_per_rev / screw_pitch_mm; // step / mm
float v_nom_steps_s = v_nom_mm_s * steps_per_mm;     // step / s

unsigned long start_time = millis();   // Memorizza il tempo di inizio del movimento.

// Variabili per il controllo del tempo e della velocit .
unsigned long t_prev_step = micros();
float current_speed = 0.0;

// Ciclo infinito per il movimento del motore.
while (true)
{
 // Calcola il tempo attuale e il tempo trascorso in secondi.
  unsigned long t_now = millis();
  float t = (t_now - start_time) / 1000.0; // seconds

  // Determina la fase e la velocit  attuale del motore.
  if (t < Dt_I)
  {
    // Fase di accelerazione (Ramp-Up).
    current_speed = v_nom_steps_s * (t / Dt_I);
  }
  else if (t < Dt_I + Dt_II)
  {
    // Fase di velocit  costante.
    current_speed = v_nom_steps_s;
  }
  else if (t < Dt_I + Dt_II + Dt_III)
  {
    // Fase di decelerazione (Ramp-Down).
    float t_dec = t - Dt_I - Dt_II;
    current_speed = v_nom_steps_s * (1.0 - t_dec / Dt_III);
  }
  else
  {
    // Fine del movimento.
    break;
  }

  // Calcola l'intervallo di tempo tra i passi in microsecondi.
  float step_interval_us = 1e6 / current_speed; // Microseconds between steps

  // Controlla se    il momento di fare un passo.
  if ((micros() - t_prev_step) >= step_interval_us)
  {
    // Muove il motore.
    stepMotor(motor);
```

```
222       // Aggiorna il tempo dell'ultimo passo.
223       t_prev_step = micros();
224     }
225   }
226
227
228   // Disable Motor
229   digitalWrite(EN_PINs[motor], HIGH);
230 }
231
232 // -----------------------------
233 // go_to_position(target_mm) : Moves a stepper connected to STEP_PIN up to
      target_mm millimeters from current_postion_mm
234 void go_to_position(float target_mm, int motor)
235 {
236   // Enable Motor
237   digitalWrite(EN_PINs[motor], LOW);
238
239   // Compute nominal steps per second
240   // Calcola i passi nominali per secondo.
241   // 'steps_per_mm' rappresenta il numero di passi per millimetro.
242   // 'distance_mm' rappresenta la distanza da percorrere in millimetri.
243   float steps_per_mm = steps_per_rev / screw_pitch_mm; // step / mm
244   float distance_mm = target_mm - current_position_mm; // mm
245
246   // Checks sign of the distance to travel
247   // Se la distanza   positiva, il motore deve muovere in avanti;
      altrimenti, indietro.
248   bool forward = distance_mm >= 0; // boolean
249   float total_distance_mm = abs(distance_mm);
250   unsigned long total_steps = total_distance_mm * steps_per_mm; // Numero
      totale di passi da eseguire.
251
252   // Imposta la direzione del motore.
253   digitalWrite(DIR_PINs[motor], forward ? HIGH : LOW);
254
255   // Calcola la velocit  nominale in passi al secondo.
256   float v_nom_steps_s = v_nom_mm_s * steps_per_mm;
257   float step_interval_us = 1e6 / v_nom_steps_s; // Intervallo di tempo tra
      i passi in microsecondi.
258
259
260   unsigned long step_count = 0; // Contatore dei passi eseguiti.
261   unsigned long t_prev = micros(); // Memorizza il tempo dell'ultimo passo
      .
262
263   // Ciclo fino a quando non sono stati eseguiti tutti i passi necessari.
264   while (step_count < total_steps)
265   {
266     if ((micros() - t_prev) >= step_interval_us)
267     {
268       stepMotor(motor);
269       step_count++;
270       t_prev = micros();
271     }
```

```
  }

  // Aggiorna la posizione corrente dell'attuatore.
  current_position_mm = target_mm; // mm


  // Disable Motor
  digitalWrite(EN_PINs[motor], HIGH);
}


// ------------------------------
// go_to_angle(target_deg) : Moves the stepper (motor) to target_mm millimeters from current_postion_mm
void go_to_angle(float target_deg, int motor)
{
  // Enable Motor
  digitalWrite(EN_PINs[motor], LOW);

  // Calcola il numero di passi per grado.
// 'steps_per_degree' rappresenta il numero di passi per grado.
// 'angle_diff' rappresenta la differenza angolare da percorrere in gradi.
  float steps_per_degree = steps_per_rev / 360.0f; // step / deg
  float angle_diff = target_deg - current_angle_deg; // deg

   // Controlla la direzione del movimento.
   // Se la differenza angolare   positiva, il motore deve muovere in avanti; altrimenti, indietro.
  bool forward = angle_diff >= 0;
  float total_angle = abs(angle_diff);
  float total_steps = total_angle * steps_per_degree;
// Serial.println("!!! DEBUG : Total Steps go_to_angle: " + String(total_steps) + " steps.");

  // Imposta la direzione del motore.
  digitalWrite(DIR_PINs[motor], forward ? HIGH : LOW);

  // Calcola la velocit  nominale in passi al secondo.
  float v_nom_steps_s = v_nom_mm_s * (steps_per_rev / screw_pitch_mm); // convert to steps/s if desired
  float step_interval_us = 1e6 / v_nom_steps_s;  // Intervallo di tempo tra i passi in microsecondi.

  unsigned long step_count = 0; // Contatore dei passi eseguiti.
  unsigned long t_prev = micros(); // Memorizza il tempo dell'ultimo passo


  // Ciclo fino a quando non sono stati eseguiti tutti i passi necessari.
  while (step_count < total_steps)
  {
    if ((micros() - t_prev) >= step_interval_us)
    {
      stepMotor(motor);
      step_count++;
      t_prev = micros();
```

```
322     }
323   }
324
325   // Aggiorna l'angolo corrente del motore.
326   current_angle_deg = target_deg;
327
328
329   // Disable Motor
330   digitalWrite(EN_PINs[motor], HIGH);
331
332 }
333
334
335 bool waitForUserResponse(String prompt,long timeout_ms)
336 {
337     // Ask the user a yes/no question and wait for input with timeout
338     // Restituisce true se l'utente risponde 'y' o 'Y', false se risponde
    'n' o 'N',
339     // e assume false se scade il timeout.
340     Serial.println(prompt);
341     Serial.print("Enter 'y' or 'n' within ");
342     Serial.print(timeout_ms / 1000);
343     Serial.println(" seconds:");
344
345     unsigned long start_time = millis();    // Memorizza il tempo di
    inizio.
346
347     // Ciclo fino a quando non scade il timeout.
348     while (millis() - start_time < timeout_ms) {
349                 // Controlla se ci sono dati disponibili sulla seriale.
350         if (Serial.available()) {
351         char response = Serial.read(); // Legge la risposta dell'utente.
352             // Se la risposta    'y' o 'Y', restituisce true.
353         if (response == 'y' || response == 'Y') {
354             return true;
355         } else if (response == 'n' || response == 'N') {
356             return false;
357         }
358         }
359     }
360
361     // If timeout expires, assume "no"
362     Serial.println("No response received. Assuming motor is homed.");
363     return false;
364 }
365
366
367 void setup()
368 {
369     // Inizializza la seriale - OK
370     Serial.begin(115200);
371     Serial.setTimeout(10); // Era inizialmente 50; da ridurre se va a
    scatti, da aumentare se si perde dei pezzi nella comunicazione.
372
373
```

```
374
375     // Assegnazione PIN
376     for (int stepper_index = 1; stepper_index < 4; stepper_index++)
377     {
378         pinMode(EN_PINs[stepper_index],  OUTPUT);
379         pinMode(DIR_PINs[stepper_index], OUTPUT);
380         // Motore disabilitato all'avvio (HIGH di solito    disable per i
    driver TB6600/A4988)
381         digitalWrite(EN_PINs[stepper_index], HIGH);
382     }
383
384     // PER DEBUG!!!!
385     // Configura il LED integrato come uscita
386   pinMode(LED_BUILTIN, OUTPUT);
387
388 }
389
390
391
392 void loop() {
393   // --- STATO A: RICEZIONE PARAMETRI ---
394   // Si attiva all'avvio e termina quando riceve la stringa "INITIAL_PAR"
395   if (!parameters_confirmed) {
396     if (Serial.available() > 0) {
397       char identificatore = Serial.peek();
398
399       if (identificatore == '?') {
400         Serial.read();
401         Serial.println(FIRMWARE_ID);
402       }
403       else if (identificatore == 'I') {
404         if (Serial.find("INITIAL_PAR:")) {
405           // Lettura sequenziale dei 4 parametri da Simulink
406           initial_gamma_ang        = Serial.parseFloat();
407           initial_delta_mm         = Serial.parseFloat();
408           speed_max_user_input_mms = Serial.parseFloat();
409           dt                       = Serial.parseFloat(); // <--- NUOVO:
    riceve t_span
410
411           // Conferma ricezione a Simulink (aggiornata con 4 parametri)
412           for (int i = 0; i < 3; i++) {
413             Serial.print("PARAMETERS_SAVED:");
414             Serial.print(initial_gamma_ang); Serial.print(",");
415             Serial.print(initial_delta_mm); Serial.print(",");
416             Serial.print(speed_max_user_input_mms); Serial.print(",");
417             Serial.print(dt, 3); // <--- NUOVO: manda conferma del dt
    ricevuto; 3    il numero di decimali, che serve per dare accuratezza
418             Serial.println(":SAVED");
419             delay(10);
420           }
421
422           Serial.flush();
423           while(Serial.available() > 0) Serial.read(); // Pulisce il
    buffer
424
```

```
425           parameters_confirmed = true;
426           motor_index = 1;
427           current_pos = 0.0;
428         }
429       }
430       else {
431         Serial.read(); // Scarta caratteri ignoti
432       }
433     }
434 }
435
436   // --- STATO B: HOMING ---
437   // Si attiva dopo lo Stato A. Termina quando motor_index > 3 e riceve 'n
      '
438   // --- STATO B: HOMING ---
439   else if (!homed) {
440     static unsigned long lastMsg = 0;
441     static bool instructionsSent = false;
442
443     if (!instructionsSent) {
444       Serial.println("Starting homing operations:");
445       instructionsSent = true;
446     }
447
448     if (motor_index > 3) {
449       if (millis() - lastMsg > 500) {
450         Serial.println("ALL_MOTORS_HOMED");
451         lastMsg = millis();
452       }
453     }
454     else {
455       if (millis() - lastMsg > 1500) {
456         Serial.print("ASK_HOMING_MOTOR:");
457         Serial.println(motor_index);
458         lastMsg = millis();
459       }
460     }
461
462     if (Serial.available() > 0) {
463       char cmd = Serial.read();
464
465       if (cmd == 'y') {
466         float incremento = (motor_index == 3) ? 0.1 : 5.0; // Ridotto
    incremento per sicurezza
467         current_pos += incremento;
468
469         // --- FIX: Muoviamo in RELATIVO durante l'homing per evitare
    conflitti ---
470         digitalWrite(DIR_PINs[motor_index], HIGH);
471         digitalWrite(EN_PINs[motor_index], LOW);
472         for(int i=0; i<200; i++) { // Fai un numero fisso di passi per
    ogni 'y'
473           stepMotor(motor_index);
474           delayMicroseconds(800);
475         }
```

```
476
477         Serial.print("MOVING_MOTOR:");
478         Serial.println(motor_index);
479       }
480
481       else if (cmd == 'n') {
482         Serial.print("OK_MOTOR:");
483         Serial.println(motor_index);
484
485         // --- FIX CRUCIALE: Reset coordinate fisiche al termine di ogni
     motore ---
486         current_position_mm = 0.0;
487         current_angle_deg = 0.0;
488         current_pos = 0.0;
489
490         if (motor_index >= 3) {
491           homed = true;
492           for(int i=0; i<5; i++) {
493             Serial.println("ALL_MOTORS_HOMED");
494             delay(20);
495           }
496         } else {
497           motor_index++;
498         }
499         while(Serial.available() > 0) Serial.read();
500       }
501     }
502   }
503
504
505   // --- STATO C: CONFIGURAZIONE FISICA ---
506   else if (homed && !real_time_started) { // Modificato il controllo
507     delay(1000);
508
509     delta_current = initial_delta_mm;
510     gamma_current = initial_gamma_ang;
511
512     target_delta = delta_current;
513     target_gamma = gamma_current;
514     target_speed_mms = 0.0;
515     current_speed_mms = 0.0;
516
517     // Movimento fisico iniziale
518     go_to_position(delta_current, 2);
519     go_to_angle(gamma_current, 3);
520
521     // Feedback per Simulink (Punto F)
522     for(int i=0; i<3; i++) {
523         Serial.print("CONF_DELTA:");
524         Serial.println(delta_current);
525         delay(20);
526     }
527
528     for(int i=0; i<3; i++) {
529         Serial.print("CONF_GAMMA:");
```

```
530             Serial.println(gamma_current);
531             delay(20);
532         }
533 
534         for(int i=0; i<3; i++) {
535             Serial.println("SYSTEM_READY_FOR_STEP_D");
536             delay(20);
537         }
538 
539         delay(100);
540 
541         // --- SALTO DIRETTO AL REAL TIME ---
542         real_time_started = true;
543         last_step_time_motor1 = micros();
544         last_ramp_calc = millis();
545 
546         Serial.println("DEBUG: Entering Real-Time Loop D immediately.");
547 }
548 
549 
550 
551 // --- STATO D: REAL TIME LOOP ---
552 else if (real_time_started) {
553 
554         // 1. RICEZIONE DATI DA SIMULINK (Ogni 0.5s circa)
555         if (Serial.available() > 0) {
556             if (Serial.peek() == '<') {
557                 Serial.read(); // Rimuove '<'
558 
559                 float new_v   = Serial.parseFloat(); // Speed obiettivo, mm/s
560                 float d_gamma = Serial.parseFloat(); // deg
561                 float d_delta = Serial.parseFloat(); // mm
562 
563                 while (Serial.available() > 0 && Serial.read() != '>');
564 
565                 target_speed_mms = new_v;
566 
567                 // Calcolo accelerazione necessaria per il salto in 0.5s (
    Punto D)
568                 // v_final = v_curr + a * 0.5 -> a = delta_v / 0.5
569                 float req_a = abs(target_speed_mms - current_speed_mms) / 0.5;
570                 active_accel = min(req_a, acceleration_mms2); // Saturazione
    ad acc. massima
571 
572                 // Applica variazioni Gamma e Delta (come prima)
573                 if (abs(d_gamma) > 0.005) {
574                     current_angle_deg += d_gamma; // Aggiornamento stato
575                     go_to_angle(current_angle_deg, 3);
576                 }
577                 if (abs(d_delta) > 0.005) {
578                     current_position_mm += d_delta; // Aggiornamento stato
579                     go_to_position(current_position_mm, 2);
580                 }
581             } else {
582                 Serial.read();
```

```
583         }
584     }
585 
586     // 2. GESTIONE RAMPA E INTEGRAZIONE POSIZIONE (Ogni 10ms)
587     unsigned long current_millis = millis();
588     if (current_millis - last_ramp_calc >= 10) {
589 
590         float dt_int = (current_millis - last_ramp_calc) / 1000.0; //
    Tipicamente 0.01s
591         float v_prev = current_speed_mms;
592         float v_new = v_prev;
593 
594         // Logica di accelerazione/decelerazione
595         if (v_prev < target_speed_mms) {
596             v_new = v_prev + active_accel * dt_int;
597             if (v_new > target_speed_mms) v_new = target_speed_mms;
598         }
599         else if (v_prev > target_speed_mms) {
600             v_new = v_prev - active_accel * dt_int;
601             if (v_new < target_speed_mms) v_new = target_speed_mms;
602         }
603 
604 
605         current_speed_mms = v_new; // Aggiorna la velocit  corrente
606         last_ramp_calc = current_millis;
607     }
608 
609     // 3. GESTIONE PASSI MOTORE 1 (Alta priorit )
610     if (abs(current_speed_mms) > 0.1) {
611        float steps_per_mm = steps_per_rev / screw_pitch_mm;
612        float steps_per_sec = abs(current_speed_mms) * steps_per_mm;
613        unsigned long interval_us = 1000000.0 / steps_per_sec;
614 
615        if (micros() - last_step_time_motor1 >= interval_us) {
616           digitalWrite(DIR_PINs[1], current_speed_mms > 0 ? HIGH : LOW);
617 
618           digitalWrite(STEP_PINs[1], HIGH);
619           delayMicroseconds(10);
620           digitalWrite(STEP_PINs[1], LOW);
621 
622           digitalWrite(EN_PINs[1], LOW);
623           last_step_time_motor1 = micros();
624        }
625     }
626 }
627 
628 }
```

Bibliography

[1] T. Binetti, “Preliminary study of a docking system for a lunar pressurized rover”, M.S. thesis, Politecnico di Torino, 2025.

[2] M. Grieves, *Virtually Perfect: Driving Innovative and Lean Products through Product Lifecycle Management*. 2011, p. 133, ISBN: 0982138008.

[3] M. Shafto et al., “Draft modeling, simulation, information technology & processing roadmap”, *Technology area*, vol. 11, pp. 1–32, 2010.

[4] E. Glaessgen and D. Stargel, “The digital twin paradigm for future nasa and us air force vehicles”, in *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*, 2012, p. 1818. DOI: 10.2514/6.2012-1818.

[5] F. Tao, B. Xiao, Q. Qi, J. Cheng, and P. Ji, “Digital twin modeling”, *Journal of Manufacturing Systems*, vol. 64, pp. 372–389, 2022. DOI: https://doi.org/10.1016/j.jmsy.2022.06.015.

[6] W. Liu, M. Wu, G. Wan, and M. Xu, “Digital Twin of Space Environment: Development, Challenges, Applications, and Future Outlook”, *Remote Sensing*, vol. 16, no. 16, 2024. DOI: 10.3390/rs16163023.

[7] B. R. Barricelli, E. Casiraghi, and D. Fogli, “A survey on digital twin: Definitions, characteristics, applications, and design implications”, *IEEE access*, vol. 7, pp. 167 653–167 671, 2019. DOI: https://doi.org/10.1109/ACCESS.2019.2953499.

[8] A. Fuller, Z. Fan, C. Day, and C. Barlow, “Digital twin: Enabling technologies, challenges and open research”, *IEEE Access*, vol. 8, pp. 108 952–108 971, 2020. DOI: 10.1109/ACCESS.2020.2998358.

[9] M. Grieves, “Digital Twin: Manufacturing Excellence through Virtual Factory Replication”, 2015.

[10] M. Grieves and J. Vickers, “Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems”, in 2017, pp. 85–113, ISBN: 978-3-319-38754-3. DOI: 10.1007/978-3-319-38756-7_4.

[11] D. Wagg, K. Worden, R. Barthorpe, and P. Gardner, "Digital twins: State-of-the-art and future directions for modeling and simulation in engineering dynamics applications", *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering*, vol. 6, no. 3, p. 030 901, 2020. DOI: 10.1115/1.4046739.

[12] F. Tao, M. Zhang, Y. Liu, and A. Nee, "Digital twin driven prognostics and health management for complex equipment", *CIRP Annals*, vol. 67, no. 1, pp. 169–172, 2018. DOI: 10.1016/j.cirp.2018.04.055.

[13] J Le Moigne et al., "NASA Earth System Digital Twins (ESDT) Prototypes", in *2024 DestinE 3rd User Exchange Workshop*, 2024.

[14] K. Ashton et al., "That 'internet of things' thing", *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[15] Wikipedia, *Ipv4 address exhaustion*, https://en.wikipedia.org/wiki/IPv4_address_exhaustion, Accessed: 2025-10-14.

[16] R. E. Arvidson et al., "Mars science laboratory curiosity rover megaripple crossings up to sol 710 in gale crater", *Journal of Field Robotics*, vol. 34, no. 3, pp. 495–518, 2017. DOI: 10.1002/rob.21647.

[17] NASA, *Twin of nasa's perseverance mars rover begins terrain tests*, https://www.nasa.gov/centers-and-facilities/jpl/twin-of-nasas-perseverance-mars-rover-begins-terrain-tests/, Accessed: 2025-09-11.

[18] NASA, *Water walk: Training for a hubble mission (1999)*, https://science.nasa.gov/asset/hubble/water-walk-training-for-a-hubble-mission-1999/, Accessed: 2025-09-11.

[19] X. O'Keefe, K. McCutchan, A. Muniz, J. Burns, and D. Szafir, "Practice makes perfect: A study of digital twin technology for assembly and problem-solving using lunar surface telerobotics", *Advances in Space Research*, vol. 76, no. 3, pp. 1550–1562, 2025. DOI: https://doi.org/10.1016/j.asr.2025.05.048.

[20] M. U. Kim, "A survey on digital twin in aerospace in the new space era", in *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*, IEEE, 2022, pp. 1735–1737. DOI: 10.1109/ICTC55196.2022.9952929.

[21] L. Pinello, L. Brancato, M. Giglio, F. Cadini, and G. F. De Luca, "Enhancing planetary exploration through digital twins: A tool for virtual prototyping and hums design", *Aerospace*, vol. 11, no. 1, p. 73, 2024. DOI: 10.3390/aerospace11010073.

[22] B.-H. Chen, P. Negrut, T. Liang, N. Batagoda, H. Zhang, and D. Negrut, "POLAR3D: Augmenting NASA's POLAR Dataset for Data-Driven Lunar Perception and Rover Simulation", *arXiv*, 2023. DOI: 10.48550/arXiv.2309.12397.

[23] T. Nilsson et al., "Using virtual reality to shape humanity's return to the moon: Key takeaways from a design study", in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–16. DOI: 10.1145/3544548.3580718.

[24] R. Loftin and P. Kenney, "Training the hubble space telescope flight team", *IEEE Computer Graphics and Applications*, vol. 15, no. 5, pp. 31–37, 1995. DOI: 10.1109/38.403825.

[25] S. Parkes, I. Martin, M. Dunstan, and D Matthews, "Planet surface simulation with pangu", in *Space ops 2004 conference*, 2004, p. 389.

[26] W. Liu, G. Wan, J. Liu, and D. Cong, "Path Planning for Lunar Rovers in Dynamic Environments: An Autonomous Navigation Framework Enhanced by Digital Twin-Based A*-D3QN", *Aerospace*, vol. 12, no. 6, 2025. DOI: 10.3390/aerospace12060517.

[27] J. Carrillo et al., "Enabling Interoperable Digital Twins for Collaborative Lunar Exploration", in *2025 IEEE Aerospace Conference*, 2025, pp. 1–10. DOI: 10.1109/AERO63441.2025.11068549.

[28] CIMdata, *The mathworks introduces simscape for modeling and simulating multidomain physical systems*, https://www.cimdata.com/newsletter/2008/9/03/09.03.10.htm, Accessed: 2025-09-01.

[29] PROJECTCHRONO, *What is projectchrono?*, https://projectchrono.org/about/, Accessed: 2025-09-01.

[30] cgchannel, *Nvidia launches omniverse: Still free to individual artists*, https://www.cgchannel.com/2022/01/nvidia-launches-omniverse-still-free-to-individual-artists/, Accessed: 2025-09-01.

[31] E. Chow et al., "Collaborative moonwalkers", in *2024 IEEE Aerospace Conference*, IEEE, 2024, pp. 1–15.

[32] N. Gratius, Y. Hou, M. Bergés, and B. Akinci, "Lessons learned on the implementation of probabilistic graphical model-based digital twins: A space habitat study", *Journal of Space Safety Engineering*, vol. 10, no. 2, pp. 172–181, 2023. DOI: 10.1016/j.jsse.2023.04.001.

[33] A. Jakeman, R. Letcher, and J. Norton, "Ten iterative steps in development and evaluation of environmental models", *Environmental Modelling & Software*, vol. 21, no. 5, pp. 602–614, 2006, ISSN: 1364-8152. DOI: 10.1016/j.envsoft.2006.01.004.

[34] C. A. George, "Deep-space habitat autonomy supported by hierarchical digital twin architecture", M.S. thesis, University of California, Davis, 2023.

[35] E. Astronautica, *Gemini 7*, http://www.astronautix.com/g/gemini7.html, Accessed: 2025-10-17.

[36] W. Fehse, *Automated rendezvous and docking of spacecraft*. Cambridge university press, 2003, vol. 16.

[37] S. R. Donahoe, "International docking system standard (idss) interface definition document (idd) revision f", Tech. Rep., 2022.

[38] T. MOHTAR EIZAGA, "Design and modeling of a space docking mechanism for cooperative on-orbit servicing", 2018.

[39] H. W. Dotts, R. K. Nolting, W. F. Hoyler, J. R. Havey, T. F. Carter Jr, and R. T. Johnson, “Operational characteristics of the docked configuration”, *NASA Special Publication*, vol. 138, p. 41, 1967.

[40] V. Syromyatnikov, “Docking devices for soyuz-type spacecraft”, in *6th Aerospace Mechanisms Symposium*, National Aeronautics and Space Administration, 1972.

[41] E. C. Ezell and L. N. Ezell, *The partnership: A NASA history of the Apollo-Soyuz test project*. Courier Corporation, 2013.

[42] W. Swan Jr, “Apollo-soyuz test project docking system”, in *JPL 10th Aerospace Mech. Symp.*, 1976.

[43] Wikipedia, *Common berthing mechanism*, https://en.wikipedia.org/wiki/Common_Berthing_Mechanism, Accessed: 2025-11-19.

[44] M. M. Cohen, “Pressurized rover airlocks”, SAE Technical Paper, Tech. Rep., 2000.

[45] R. L. Howard, “A multi-gravity docking and utilities transfer system for a common habitat architecture”, in *ASCEND 2021*, 2021, p. 4062. DOI: doi.org/10.2514/6.2021-4062.

[46] A. F. Abercromby, M. L. Gernhardt, and H. Litaker, *Desert Research and Technology Studies (DRATS) 2009: A 14-day evaluation of the Space Exploration Vehicle prototype in a lunar analog environment*. National Aeronautics and Space Administration, Lyndon B. Johnson Space Center, 2012.

[47] G. Heiken, D. Vaniman, and B. M. French, *Lunar sourcebook: A user’s guide to the Moon*. Cup Archive, 1991.

[48] P. Lucey et al., “Understanding the lunar surface and space-moon interactions”, *Reviews in mineralogy and geochemistry*, vol. 60, no. 1, pp. 83–219, 2006.

[49] A. MotionView, *Soft soil tire model*, https://2022.help.altair.com/2022/hwdesktop/mv/topics/motionview/soft_soil_tire_r.htm, Accessed: 2025-11-6.

[50] U. S. Tech, *1.5 million dollars per kg: Intuitive machines will deliver cargo to the south pole of the moon on nasa’s order*, https://universemagazine.com/en/1-5-million-dollars-per-kg-intuitive-machines-will-deliver-cargo-to-the-south-pole-of-the-moon-on-nasas-order/, Accessed: 2025-11-6.

[51] P. di Torino, *Space it up: The consortium for italian space excellence is born (traduced from italian)*, https://www.polito.it/ateneo/comunicazione-e-ufficio-stampa/poliflash/space-it-up-nasce-la-societa-consortile-per-l-eccellenza, Accessed: 2025-11-21.

[52] S. I. Up, *Space it up homepage*, https://spaceitup.it/, Accessed: 2025-12-7.

[53] NASA, *Appendix c: How to write a good requirement— checklist*, https://www.nasa.gov/reference/appendix-c-how-to-write-a-good-requirement/, Accessed: 2025-12-4.

[54] M. Segovia and J. Garcia-Alfaro, “Design, modeling and implementation of digital twins”, *Sensors*, vol. 22, no. 14, 2022. DOI: 10.3390/s22145396.

[55] R. Alizadeh, J. K. Allen, and F. Mistree, "Managing computational complexity using surrogate models: A critical review", *Research in Engineering Design*, vol. 31, no. 3, pp. 275–298, 2020.

[56] A. Rasheed, O. San, and T. Kvamsdal, "Digital twin: Values, challenges and enablers from a modeling perspective", *IEEE Access*, vol. 8, pp. 21 980–22 012, 2020. DOI: 10.1109/ACCESS.2020.2970143.

[57] P. Albertos and G. C. Goodwin, "Virtual sensors for control applications", *Annual Reviews in Control*, vol. 26, no. 1, pp. 101–112, 2002, ISSN: 1367-5788. DOI: 10.1016/S1367-5788(02)80018-9.

[58] D. Martin, N. Kühl, and G. Satzger, "Virtual sensors", *Business & Information Systems Engineering*, vol. 63, no. 3, pp. 315–323, 2021. DOI: 10.1007/s12599-021-00689-w.

[59] L. Liu, S. M. Kuo, and M. Zhou, "Virtual sensing techniques and their applications", in *2009 International Conference on Networking, Sensing and Control*, 2009, pp. 31–36. DOI: 10.1109/ICNSC.2009.4919241.

[60] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing", in *2010 13th international conference on network-based information systems*, IEEE, 2010, pp. 1–8.

[61] S. Kabadayi, A. Pridgen, and C. Julien, "Virtual sensors: Abstracting data from physical sensors", in *2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks(WoWMoM'06)*, 2006. DOI: 10.1109/WOWMOM.2006.115.

[62] K. Aberer, M. Hauswirth, and A. Salehi, "A middleware for fast and flexible sensor network deployment", in *Vldb*, vol. 6, 2006, p. 1215. DOI: 10.5555/1182635.1164243.

[63] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: A survey", *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016. DOI: 10.1109/JIOT.2015.2498900.

[64] Unionfab, *Pla density: A comprehensive guide*, https://www.unionfab.com/blog/2024/04/pla-densityy, Accessed: 2025-11-27.

[65] T. Letcher and M. Waytashek, "Material property testing of 3d-printed specimen in pla on an entry-level 3d printer", vol. 2, Dec. 2014. DOI: 10.1115/IMECE2014-39379.

[66] G. Torrente Prato, L. Sosa Vivas, J. Gonzalez-Delgado, H. Hernandez Silva, and H. León-Molina, "Experimental and numerical study of the orthotropic behavior of 3d printed polylactic acid by material extrusion: Part two: An analysis about poisson´s ratios", *Progress in Additive Manufacturing*, vol. 10, pp. 2335–2350, Aug. 2024. DOI: 10.1007/s40964-024-00753-3.

[67] I. K&J Magnetics, *Neodymium magnet specifications*, https://www.kjmagnetics.com/neodymium-magnet-specifications.asp?srsltid=AfmBOopJp3u-3--LEuUR_28xUXaIbuCBQhvZrc4iT4FvaHxqLIdHBjrB, Accessed: 2025-11-27.

[68] M. MAGNET, *Ndfeb magnet*, https://mcemagnet.com/products/ndfebmagnet/, Accessed: 2025-11-27.

[69] T. U. of Arizona, *Barry isolators selection guide*, https://wp.optics.arizona.edu/optomech/wp-content/uploads/sites/53/2016/08/Barry-isolators-selection-guide.pdf, Accessed: 2025-12-4.

[70] J. Giesbers, "Contact mechanics in msc adams-a technical evaluation of the contact models in multibody dynamics software msc adams", B.S. thesis, University of Twente, 2012.

[71] J. C. Helton and F. J. Davis, "Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems", *Reliability Engineering & System Safety*, vol. 81, no. 1, pp. 23–69, 2003.

[72] W.-L. Loh, "On latin hypercube sampling", *The annals of statistics*, vol. 24, no. 5, pp. 2058–2080, 1996.

[73] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: A survey", *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, ISSN: 0360-0300. DOI: 10.1145/3179993.

[74] fmi standard, *Fmi standard homepage*, https://fmi-standard.org/, Accessed: 2025-09-23.

[75] G. F. Belete, A. Voinov, and G. F. Laniak, "An overview of the model integration process: From pre-integration assessment to testing", *Environmental Modelling & Software*, vol. 87, pp. 49–63, 2017. DOI: https://doi.org/10.1016/j.envsoft.2016.10.013.